Coursework 1 PH20018 Submission

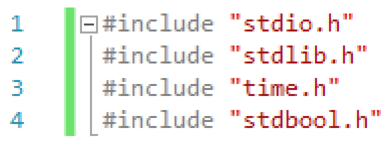
This report is the 2nd assignment for PH20018.
The report is divided in three sections, each of which solves one of the three problems related to Conductivity of disordered materials.
It is important to note that all code pertaining to this report was written in Visual Studio 2019 for visualizing purposes, but was run through command line using Min GW compiler in Windows computer.

Question 1

1. a: source1a.c program

Design presentation and discussion

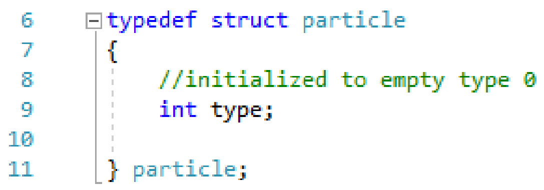
For this code the following libraries were used: standard stdio.h (standard library) , stdlib.h library (used for the function exit), the time.h library (used for the creation of random number seeds) and the stdbool.h library (used for the creation of booleans).



```
1  #include "stdio.h"
2  #include "stdlib.h"
3  #include "time.h"
4  #include "stdbool.h"
```

Figure 1: Lines 1-4 source1a.c

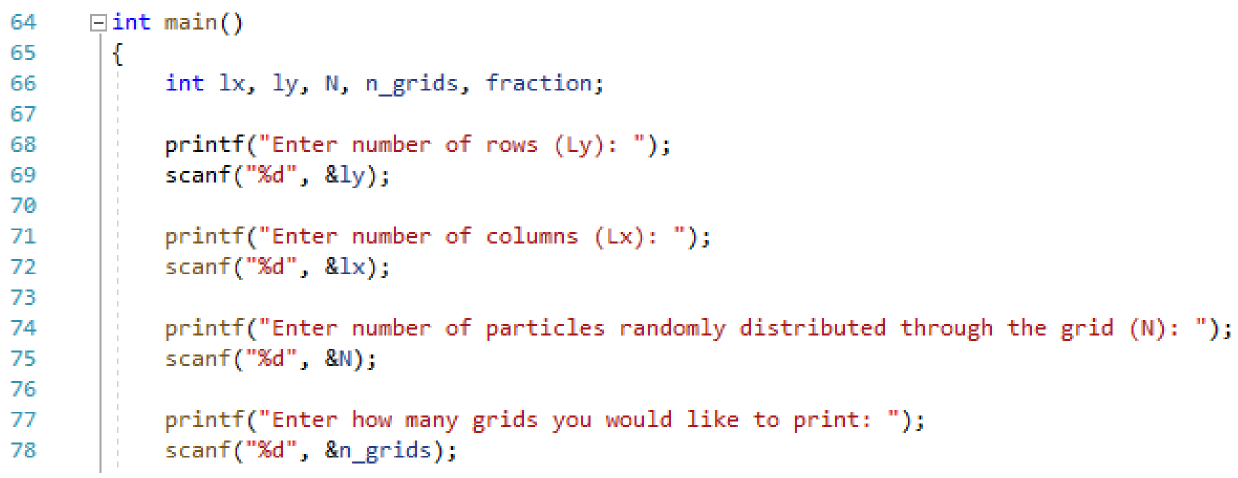
In lines **6-11** the structure particle is defined. It contains an integer structure named type, that will define wherever there is particle within the corresponding location (type =1 if it contains a particle, type =0 if the space is empty).



```
6   typedef struct particle
7   {
8       //initialized to empty type 0
9       int type;
10
11  } particle;
```

Figure 2: Lines 6-11 source1a.c

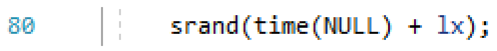
Our main function is contained between lines **64-117**. Lines **66-78** are dedicated to the input and initialization of the user input values i.e. Lx, Ly, N, and the number of grids we would like to print. Lx, Ly, N and the number of grids have an integer structure as they represent whole numbers (there is no such a thing as half a particle, half a space within the grid or half a grid).



```
64  int main()
65  {
66      int lx, ly, N, n_grids, fraction;
67
68      printf("Enter number of rows (Ly): ");
69      scanf("%d", &ly);
70
71      printf("Enter number of columns (Lx): ");
72      scanf("%d", &lx);
73
74      printf("Enter number of particles randomly distributed through the grid (N): ");
75      scanf("%d", &N);
76
77      printf("Enter how many grids you would like to print: ");
78      scanf("%d", &n_grids);
```

Figure 3: Lines 64-78 source1a.c

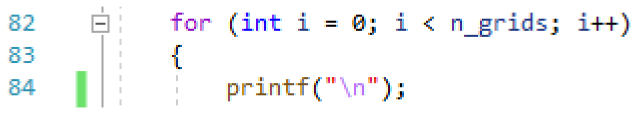
In line **80** we find a srand() function. This function contains the seed (time(NULL) + lx) that will later be used at every iteration of rand(), thus why the seed of the srand() function is based on time (to ensure the generated random numbers would be different every time the code was ran) and on a inputted number, in this case lx, to ensure even more “randomness”.



```
80      srand(time(NULL) + lx);
```

Figure 4: Line 80 source1a.c

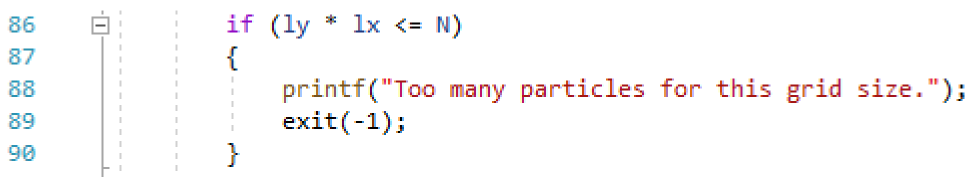
In lines **82-112** a for loop, iterates through the number of grids inputted through the command line, and thus produces as many grids as the user desires. Lines **84** and **111** both ensure there will be a printed line skip before and after every printed grid.



```
82      for (int i = 0; i < n_grids; i++)
83      {
84          printf("\n");
```

Figure 5: Lines 82-84 source1a.c

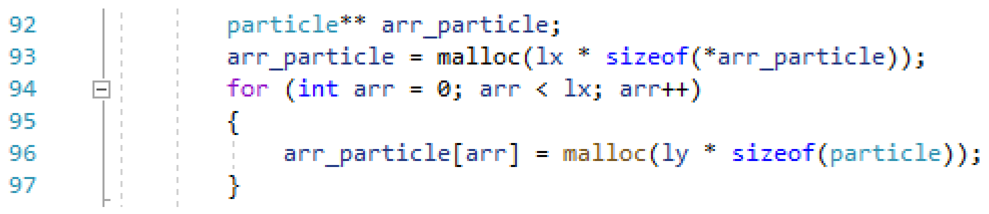
Lines **86-90** test wherever the number of particle inputted by the user can be contained within the grid. If N is superior to the number of spaces within the grid i.e. Ly*Lx, then the code prints and error message and exits.



```
86          if (ly * lx <= N)
87          {
88              printf("Too many particles for this grid size.");
89              exit(-1);
90          }
```

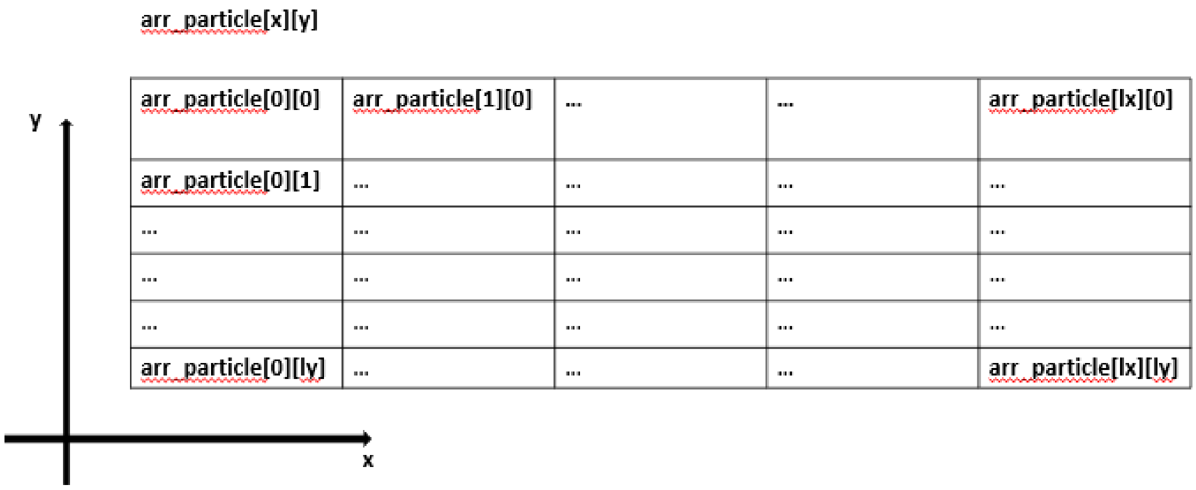
Figure 6: Lines 86-90 source1a.c

Lines **92-97** create a two dimensional particle array, that we will employ to store the information concerning our grid spaces. Each dimension of this array will define either the x or the y values regarding a position within our created grid. They will also contain, given the definition of the structure particle, information regarding the type of particle contained within those coordinates. This array is defined with the use of malloc for it’s later usage within functions, regardless of the fact it is two dimensional.



```
92          particle** arr_particle;
93          arr_particle = malloc(lx * sizeof(*arr_particle));
94          for (int arr = 0; arr < lx; arr++)
95          {
96              arr_particle[arr] = malloc(ly * sizeof(particle));
97          }
```

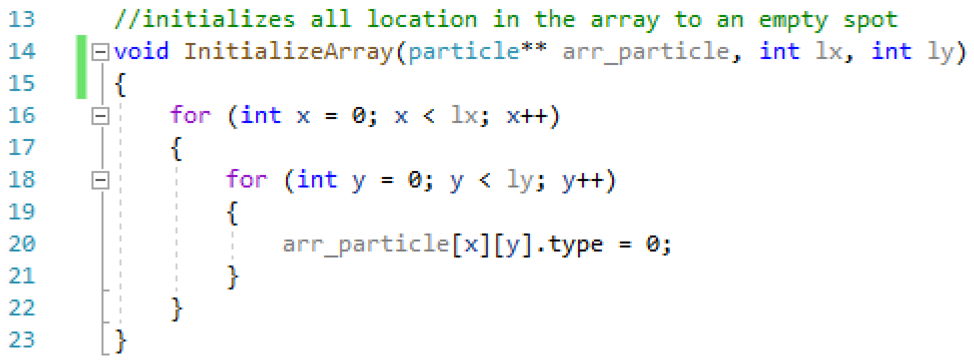
Figure 7: Lines 92-97 source1a.c



arr_particle[0][0]	arr_particle[1][0]	…	…	arr_particle[lx][0]
arr_particle[0][1]	…	…	…	…
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…
arr_particle[0][ly]	…	…	…	arr_particle[lx][ly]

Figure 8: Visual representation of the purpose of lines 92-97 source1a.c

In line **99** function "InitializeArray" is called. This function is described in lines **14-23**. The purpose of this void function is to give all the locations in our array and thus our grid an initial type = 0 which implies an empty location. It is able to do this by iterating through all possible x and y spaces through two nested for loops.

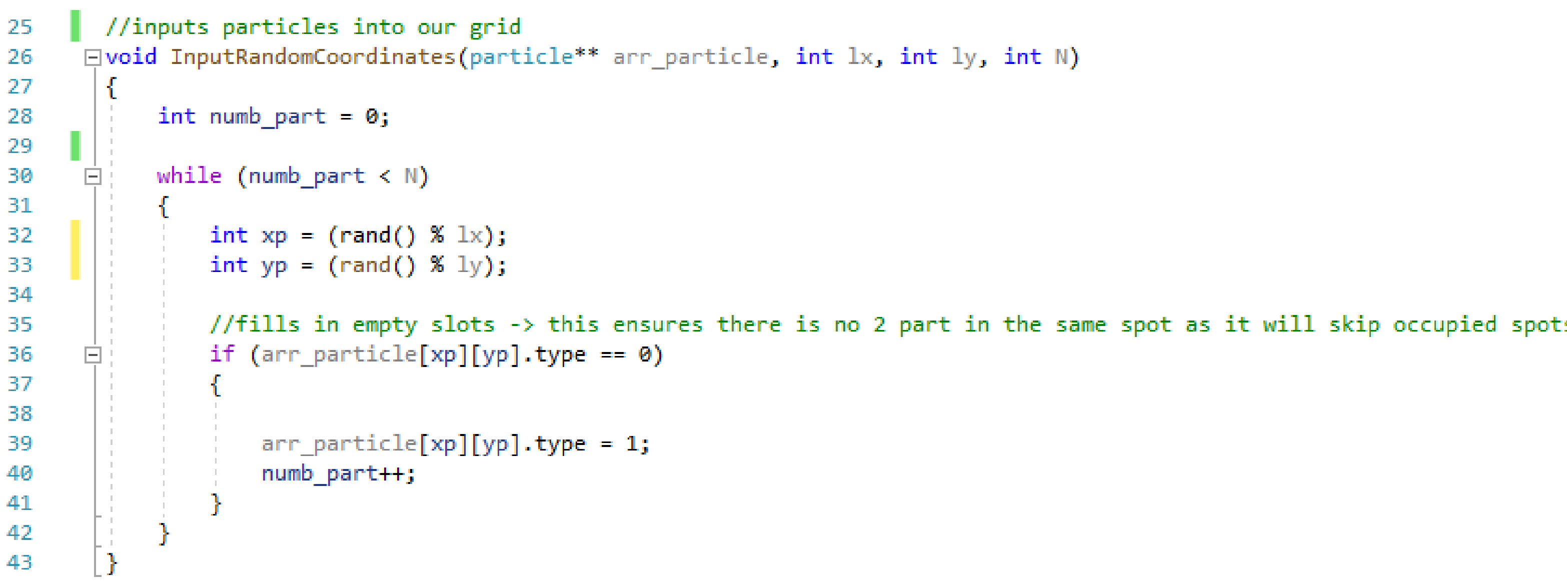


```
13      //initializes all location in the array to an empty spot
14      void InitializeArray(particle** arr_particle, int lx, int ly)
15      {
16          for (int x = 0; x < lx; x++)
17          {
18              for (int y = 0; y < ly; y++)
19              {
20                  arr_particle[x][y].type = 0;
21              }
22          }
23      }
```

Figure 9: Lines 13-23 source1a.c

In line **101** function "InputRandomCoordinates" is called. This function is described in lines **26-43**. The purpose of this void function is to fill in N random locations within our grid with particle i.e. make them type = 1.

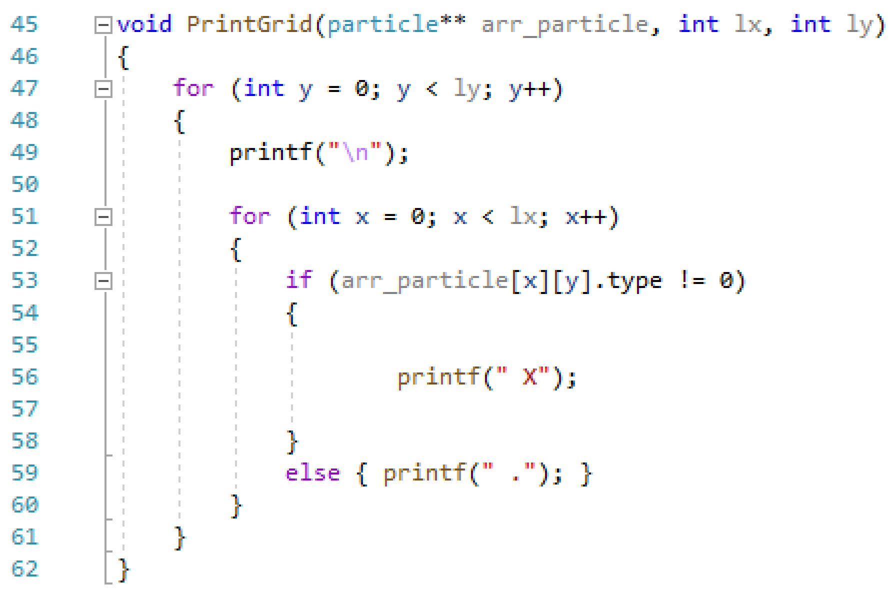
In order to do this line **27** initializes an integer num_part = 0 which describes the number of positions filled thus far with particles. In lines **30-42** a while loop will loop until num_part = N. num_part will grow by one (line **40**) if and only if a particle is added to an empty location i.e a location with type = 0. To ensure this is the case, for every iteration of the while loop, integers xp and yp are assigned a random number within the limits of the grid i.e. lx for xp and ly for yp. If the corresponding array location arr_particle[xp][yp] is empty i.e. has a type = 0, then the location is "filled" i.e it's type is changed to 1, and numb_part grows by one. This method ensures no filled location in the grid is filled more than once as it would not iterate through the if loop defined in lines **36-41**.



```
25  //inputs particles into our grid
26  void InputRandomCoordinates(particle** arr_particle, int lx, int ly, int N)
27  {
28      int numb_part = 0;
29  
30      while (numb_part < N)
31      {
32          int xp = (rand() % lx);
33          int yp = (rand() % ly);
34  
35          //fills in empty slots -> this ensures there is no 2 part in the same spot as it will skip occupied spot
36          if (arr_particle[xp][yp].type == 0)
37          {
38  
39              arr_particle[xp][yp].type = 1;
40              numb_part++;
41          }
42      }
43  }
```

Figure 10: Lines 25-43 source1a.c

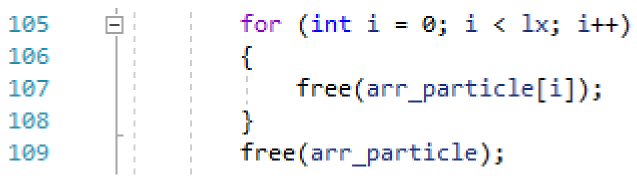
Within the main, function "PrintGrid" is called in line **103**. This function is described in lines **45-62**. The purpose of this void function is to simply print every location of our grid with a sign " X" if it's type is not zero which implies there is a particle in that location and a sign " ." if this is not the case i.e type = 0. This is done with two for loop that iterate through every x and y location within our grid and make sure to skip a line at the end of every row. The reason why the x iterating for loop is nested within the y iterating for loop instead of the other way around is linked with the way the command line prints. The command line prints from left to the right and thus this is translated as printing one row at a time. If the command line printed from top to bottom the order of the for loops would be inversed.



```
45  void PrintGrid(particle** arr_particle, int lx, int ly)
46  {
47      for (int y = 0; y < ly; y++)
48      {
49          printf("\n");
50
51          for (int x = 0; x < lx; x++)
52          {
53              if (arr_particle[x][y].type != 0)
54              {
55
56                      printf(" X");
57
58              }
59              else { printf(" ."); }
60          }
61      }
62  }
```

Figure 11: Lines 45-62 source1a.c

In lines **105-109** all created malloc pointers are freed.



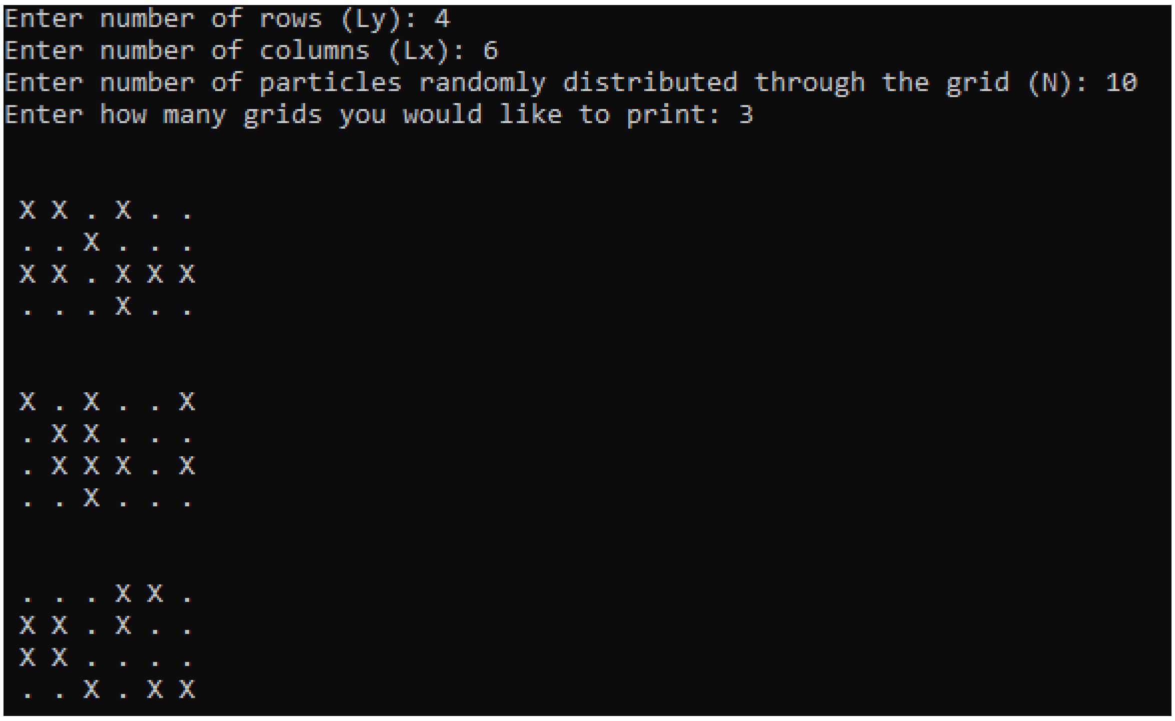
```
105         for (int i = 0; i < lx; i++)
106         {
107             free(arr_particle[i]);
108         }
109         free(arr_particle);
```

Figure 12: Lines 105-109 source1a.c

Results

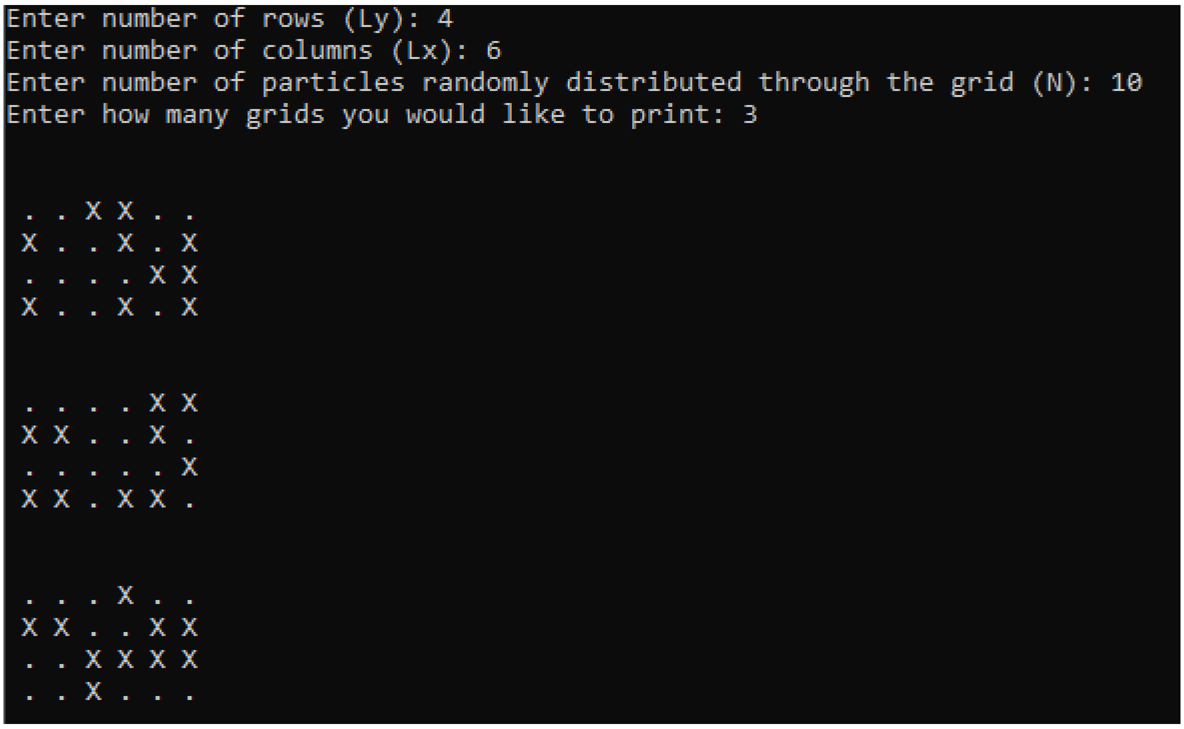
As we can observe in Figure 13 and 14 the program outputs the expected amount of grids containing the expected number of particles, which have randomly assigned locations that are different between grids and between different compilations of the code.

We thus can claim that the program source1a.c correctly answers questions 1.a)



```
Enter number of rows (Ly): 4
Enter number of columns (Lx): 6
Enter number of particles randomly distributed through the grid (N): 10
Enter how many grids you would like to print: 3


 X X . X . .
 . . X . . .
 X X . X X X
 . . . X . .


 X . X . . X
 . X X . . .
 . X X X . X
 . . X . . .


 . . . X X .
 X X . X . .
 X X . . . .
 . . X . X X
```

Figure 13: Results obtained when compiling the program source1a.c (1)



```
Enter number of rows (Ly): 4
Enter number of columns (Lx): 6
Enter number of particles randomly distributed through the grid (N): 10
Enter how many grids you would like to print: 3


 . . X X . .
 X . . X . X
 . . . . X X
 X . . X . X


 . . . . X X
 X X . . X .
 . . . . . X
 X X . X X .


 . . . X . .
 X X . . X X
 . . X X X X
 . . X . . .
```

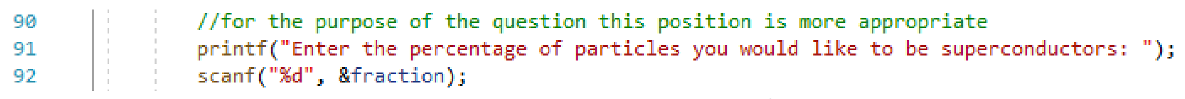
Figure 14: Results obtained when compiling the program source1a.c (2)

1. <u>b: source1b.c program</u>

<u>Design presentation and discussion</u>

Question 1b expects a modification of program source1a.c, to allow the user to input the fraction of particles, f_{SC}, that are super-conductors. Particle that are superconductor will be represented with an "X" and non-superconductor particles will be represented with a "+". In order to efficiently do this, we will start by considering that within a particle structure type = 0 corresponds to an empty grid location, type == 1 corresponds to a non-superconductor particle located in that grid location and type == -1 corresponds to a superconductor particle located in that grid location.

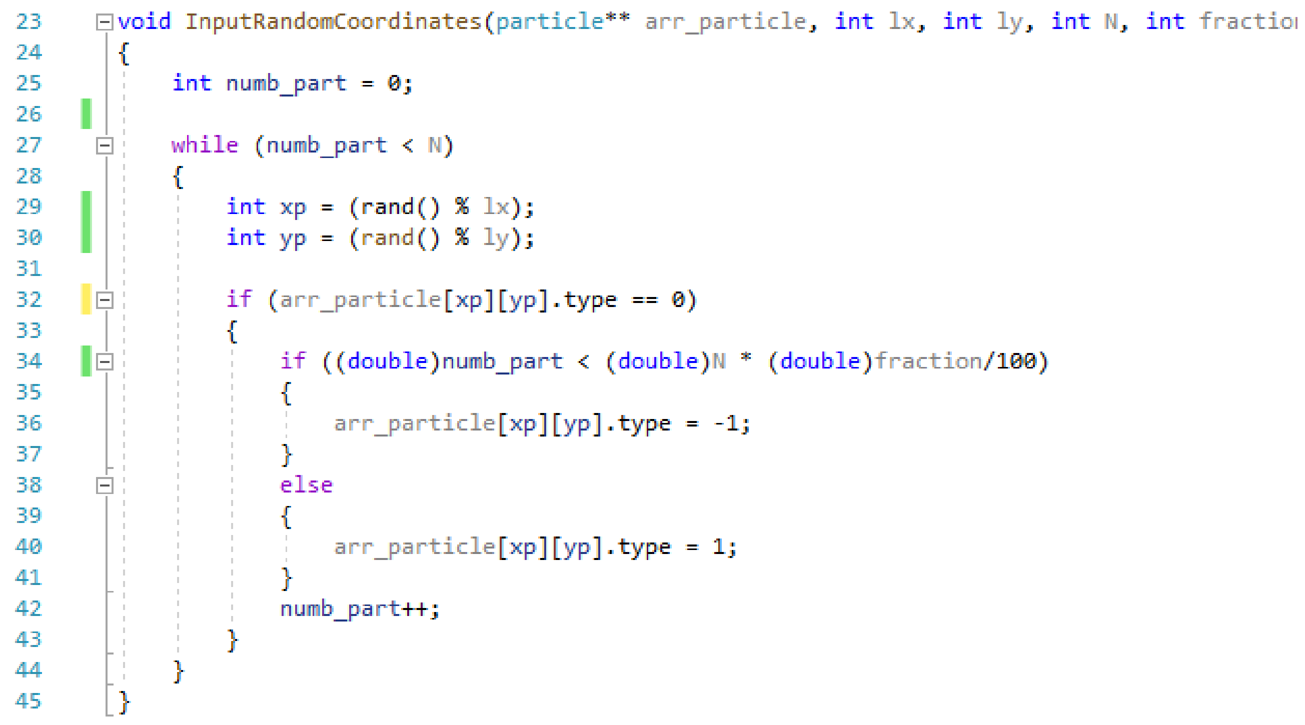
Following this we modify the function InputRandomCoordinates, present in lines **23-46**, so that once it has found an empty location in which to input it's randomly found x and y positions, it will give this one a type = -1 or a type = 1 depending on wherever the num_part of the found particle is superior or inferior to the number of particles associate with the inputted percentage in lines **91-92** (question within our main).



```
90          //for the purpose of the question this position is more appropriate
91          printf("Enter the percentage of particles you would like to be superconductors: ");
92          scanf("%d", &fraction);
```

Figure 15: Lines 90-92 source1b.c

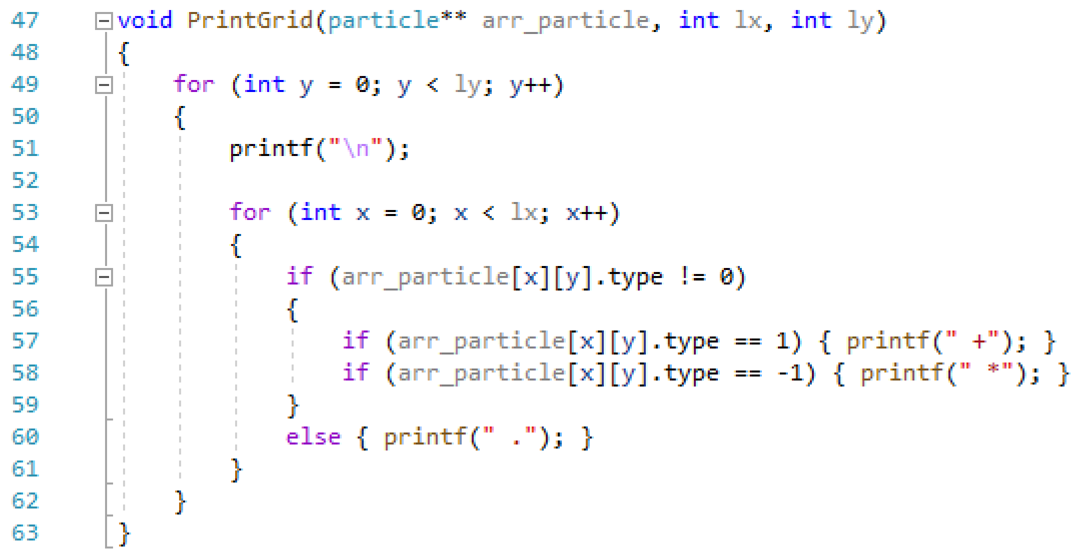
Meaning that if we have a N=16 and a f_{SC}=25, our program will make the 100* f_{SC}/N = 25/100 * 16 = 4 first randomly found particle of type =-1 i.e superconductors, and every other particle a non-superconductor.
In order for these calculations not to cause any rounding issues we cast the num_part, N and fraction inputs as doubles. The bigger or smaller than 100* f_{SC}/N condition is ensure by an if and an else loop in lines **34-41**.



```
23  void InputRandomCoordinates(particle** arr_particle, int lx, int ly, int N, int fractio
24  {
25      int numb_part = 0;
26  
27      while (numb_part < N)
28      {
29          int xp = (rand() % lx);
30          int yp = (rand() % ly);
31  
32          if (arr_particle[xp][yp].type == 0)
33          {
34              if ((double)numb_part < (double)N * (double)fraction/100)
35              {
36                  arr_particle[xp][yp].type = -1;
37              }
38              else
39              {
40                  arr_particle[xp][yp].type = 1;
41              }
42              numb_part++;
43          }
44      }
45  }
```

Figure 16: Lines 23-45 source1b.c

Our second modification within the code is present in function PrintGrid located in lines **47-63**. Once our function recognises that a locations type is not 0 and thus it is a non empty location, depending on it's type it prints " +" (type = 1, non-superconducting particle) or " *" (type=-1, superconducting particle).



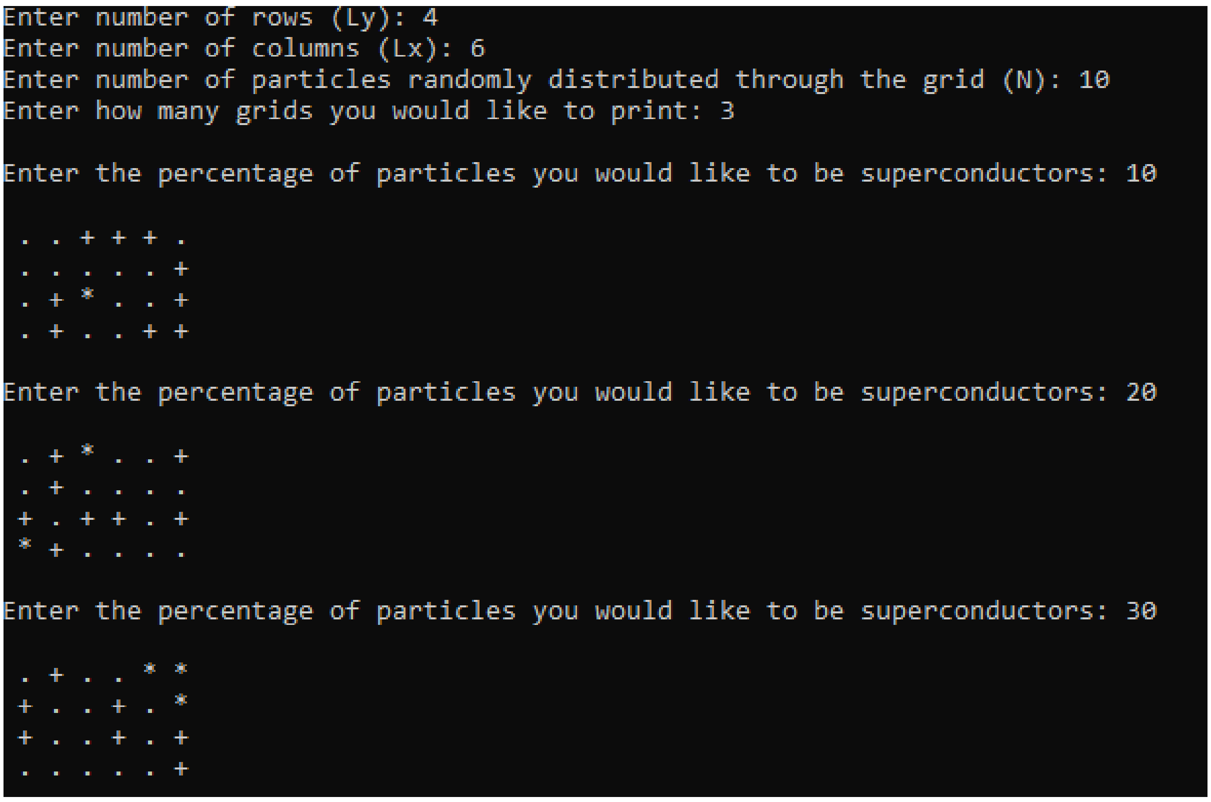
```c
47  void PrintGrid(particle** arr_particle, int lx, int ly)
48  {
49      for (int y = 0; y < ly; y++)
50      {
51          printf("\n");
52
53          for (int x = 0; x < lx; x++)
54          {
55              if (arr_particle[x][y].type != 0)
56              {
57                  if (arr_particle[x][y].type == 1) { printf(" +"); }
58                  if (arr_particle[x][y].type == -1) { printf(" *"); }
59              }
60              else { printf(" ."); }
61          }
62      }
63  }
```

Figure 17: Lines 47-63 source1b.c

Results

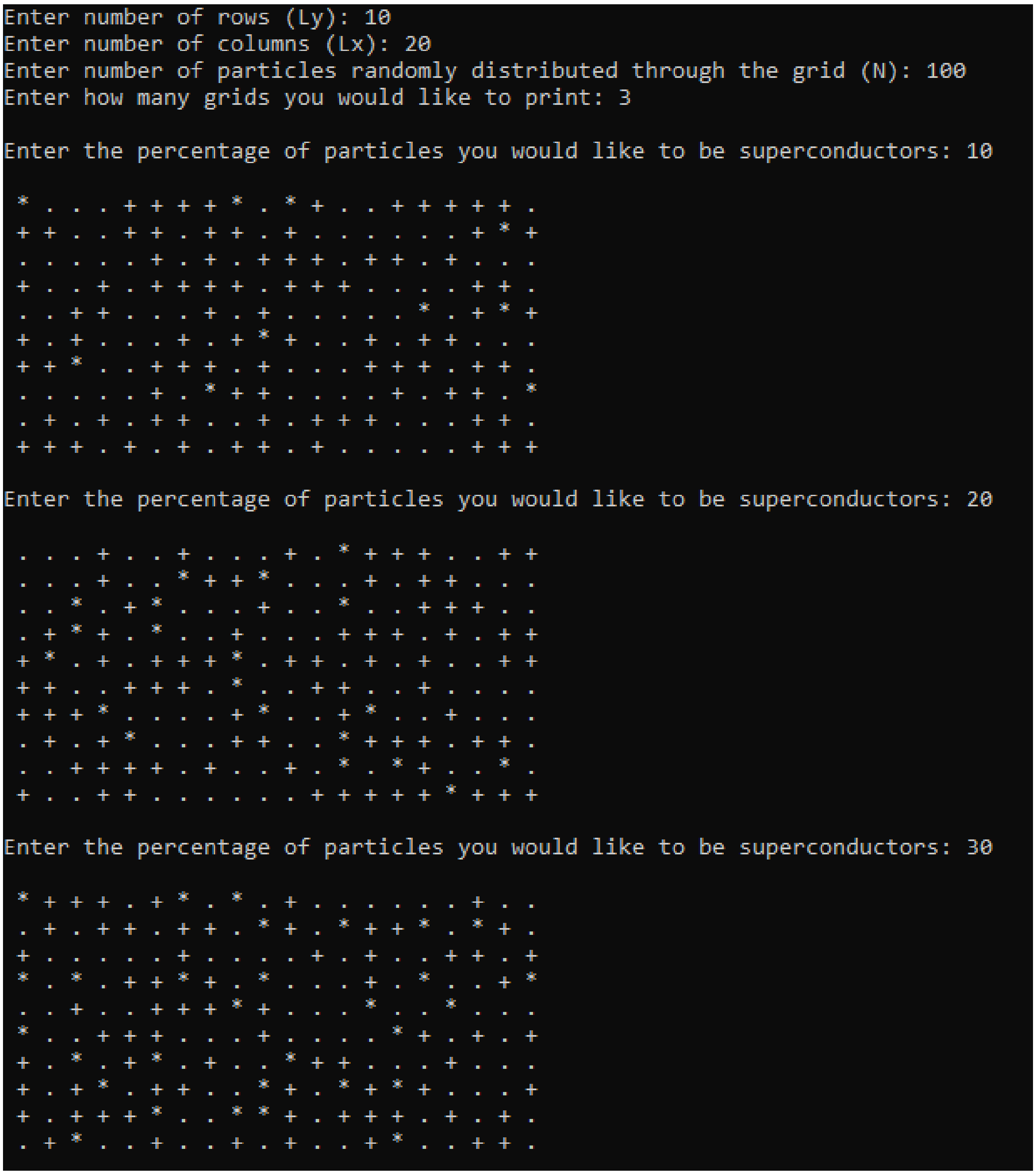
As we can observe in Figure 18 and 19 the program outputs the expected amount superconductor and non-superconductor particles regardless of lx, ly and N without loosing it's random properties.

We thus can claim that the program source1b.c correctly answers questions 1.b)



```
Enter number of rows (Ly): 4
Enter number of columns (Lx): 6
Enter number of particles randomly distributed through the grid (N): 10
Enter how many grids you would like to print: 3

Enter the percentage of particles you would like to be superconductors: 10

 . . + + + .
 . . . . . +
 . + * . . +
 . + . . + +

Enter the percentage of particles you would like to be superconductors: 20

 . + * . . +
 . + . . . .
 + . + + . +
 * + . . . .

Enter the percentage of particles you would like to be superconductors: 30

 . + . . * *
 + . . + . *
 + . . + . +
 . . . . . +
```

Figure 18: Results obtained when compiling the program source1b.c (1)



```
Enter number of rows (Ly): 10
Enter number of columns (Lx): 20
Enter number of particles randomly distributed through the grid (N): 100
Enter how many grids you would like to print: 3

Enter the percentage of particles you would like to be superconductors: 10

 * . . . + + + + * . * + . . + + + + + .
 + + . . + + . + + . + . . . . . . + * +
 . . . . . + . + . + + + . + + . + . . .
 + . . + . + + + + . + + + . . . . + + .
 . . + + . . . + . + . . . . . * . + * +
 + . + . . . + . + * + . . + . + + . . .
 + + * . . + + + . + . . . + + + . + + .
 . . . . . + . * + + . . . . + . + + . *
 . + . + . + + . . + . + + + . . . + + .
 + + + . + . + . + + . + . . . . . + + +

Enter the percentage of particles you would like to be superconductors: 20

 . . . + . . + . . . + . * + + + . . + +
 . . . + . . * + + * . . . + . + + . . .
 . . * . + * . . . + . . * . . + + + . .
 . + * + . * . . + . . . + + + . + . + +
 + * . + . + + + * . + + . + . + . . + +
 + + . . + + + . * . . + + . . + . . . .
 + + + * . . . . + * . . + * . . + . . .
 . + . + * . . . + + . . * + + + . + + .
 . . + + + + . + . . + . * . * + . . * .
 + . . + + . . . . . . + + + + + * + + +

Enter the percentage of particles you would like to be superconductors: 30

 * + + + . + * . * . + . . . . . . + . .
 . + . + + . + + . * + . * + + * . * + .
 + . . . . . + . . . . + . + . . + + . +
 * . * . + + * + . * . . . + . * . . + *
 . . + . . + + + * + . . . * . . * . . .
 * . . + + + . . . + . . . . * + . + . +
 + . * . + * . + . . * + + . . . + . . .
 + . + * . + + . . * + . * + * + . . . +
 + . + + + * . . * * + . + + + . + . + .
 . + * . . + . . + . + . . + * . . + + .
```

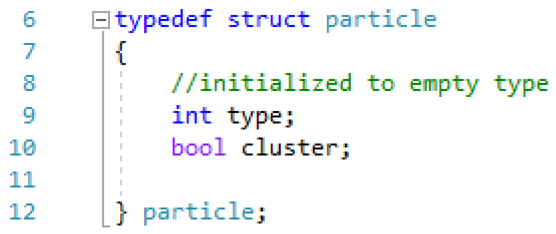
Figure 19: Results obtained when compiling the program source1b.c (2)

Question 2

2. a: source2a.c program

Design presentation and discussion

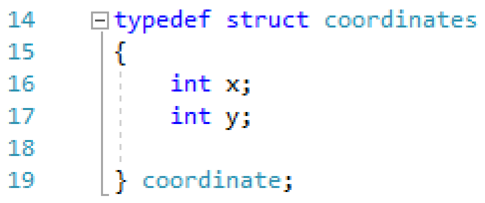
For the purpose of answering question 2.a) we add another element to our structure particle. This element is a Boolean named cluster, which will define where the array location is or isn't within the cluster containing our initial particle 0.
It is important to note that through this question we may refer to our initial particle as 0, to any particle within it's cluster as 1 and to any other particle as -1.



```
6   typedef struct particle
7   {
8       //initialized to empty type
9       int type;
10      bool cluster;
11
12  } particle;
```

Figure 20: Lines 6-12 source2a.c

In addition to this in lines **14-19** a new structure named coordinates is defined. It contains two integer types named x and y, which will later define particle coordinates.



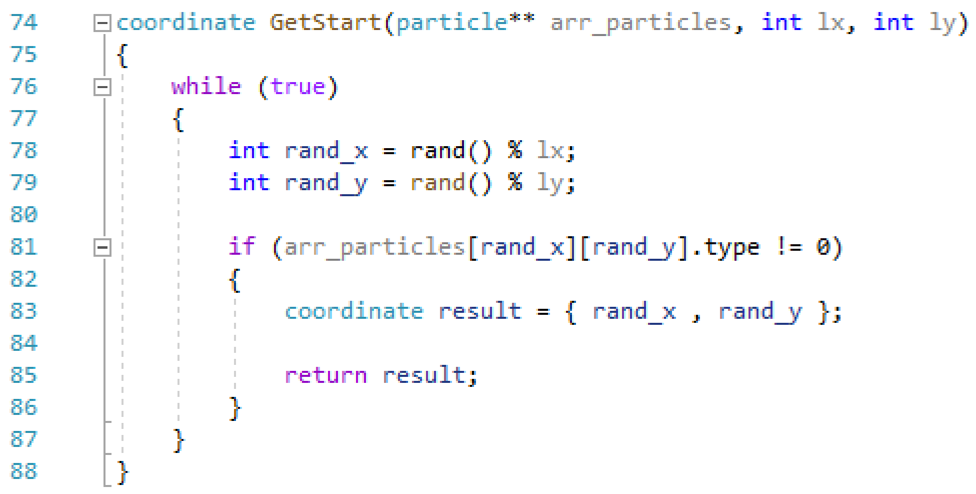
```
14  typedef struct coordinates
15  {
16      int x;
17      int y;
18
19  } coordinate;
```

Figure 21: Lines 14-19 source2a.c

Our main function now contained between lines **185-261** starts differing from the main function of program source1b.c after calling the function InputRandomCoordinates in line **225**.

Lines **227-228** define the coordinates of our particle 0, with the use of a coordinate structure and the function GetStart defined in lines **74-85**. This function, similarly to the InputRandomCoordinates function chooses one random x and one random y location, and if these correspond to an empty grid position the function returns them. If not it will keep iterating through the while loop until it finds a rand_x and a rand_y that answer the if loop's condition of corresponding to an empty space within our array and therefore our grid.

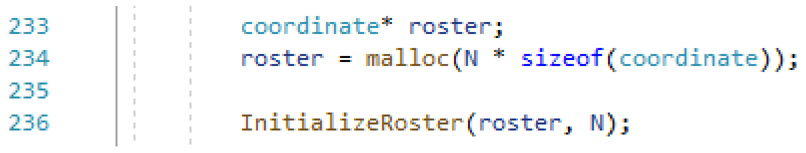
This function enables our particle 0 to be either a superconducting particle or not.



```
74  coordinate GetStart(particle** arr_particles, int lx, int ly)
75  {
76      while (true)
77      {
78          int rand_x = rand() % lx;
79          int rand_y = rand() % ly;
80  
81          if (arr_particles[rand_x][rand_y].type != 0)
82          {
83              coordinate result = { rand_x , rand_y };
84  
85              return result;
86          }
87      }
88  }
```

Figure 22: Lines 74-88 source2a.c

Similarly to lines **219-224**, lines **233-234** create a one dimensional array named rooster with a coordinate structure. Roster is defined with the use of the function malloc for it’s later usage within functions. It contains a size equivalent to N and will contain all particles within the cluster containing our initial particle i.e all particles that will have a array.cluster = true.



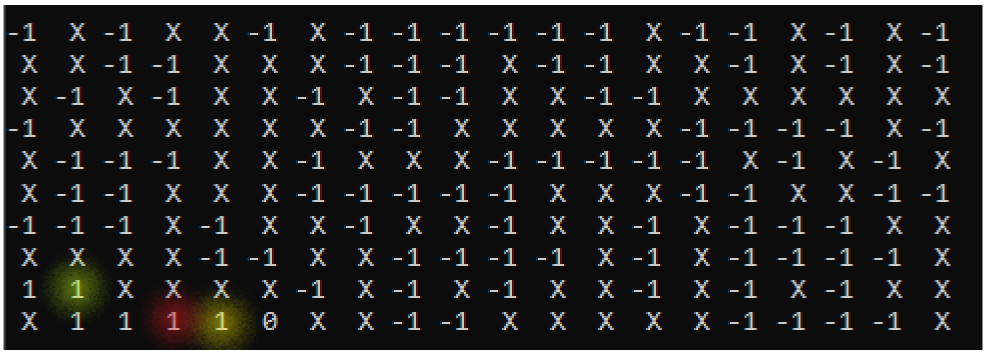
```
233             coordinate* roster;
234             roster = malloc(N * sizeof(coordinate));
235 
236             InitializeRoster(roster, N);
```

Figure 23: Lines 233-236 source2a.c

In line **236** the function InitializeRoster is called. This function is described in lines **90-97**. Similarly to the function InitializeArray, it initializes all coordinates in the roster with standard values of x and y. These standard values of x and y (in this case choosen to be x=-1 and y=-1, as neither of these values will point to a grid position) will be used to test wherever the produced roster is correct.

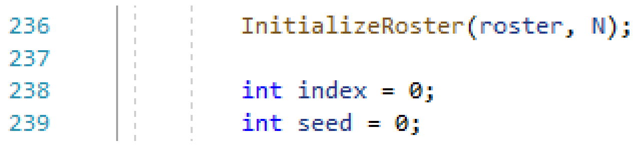
In function InitializeRoster as well as in our main (line **236**) we have an integers value named index. This value refers to the connection number of the analyzed particle with respect to our initial particle.

Meaning that in Figure 24 particle 0 would have an index = 0 as there is no jumps between itself and particle 0 (it is position 0), the particle 1 highlighted in yellow would have index = 1, the particle 1 highlighted in red would have index = 2 and the particle 1 highlighted in green would have index = 5 or 4 depending on wherever the connection has been made with the particle at index 3 or 4.



```
-1  X -1  X  X -1  X -1 -1 -1 -1 -1 -1  X -1 -1  X -1  X -1
 X  X -1 -1  X  X  X -1 -1 -1  X -1 -1  X  X -1  X -1  X -1
 X -1  X -1  X  X -1  X -1 -1  X  X -1 -1  X  X  X  X  X  X
-1  X  X  X  X  X  X -1 -1  X  X  X  X  X -1 -1 -1 -1  X -1
 X -1 -1 -1  X  X -1  X  X  X -1 -1 -1 -1 -1  X -1  X -1  X
 X -1 -1  X  X  X -1 -1 -1 -1 -1  X  X  X -1 -1  X  X -1 -1
-1 -1 -1  X -1  X  X -1  X  X -1  X  X -1  X -1 -1 -1  X  X
 X  X  X  X -1 -1  X  X -1 -1 -1 -1  X -1  X -1 -1 -1 -1  X
 1  1  X  X  X  X -1  X -1  X -1  X  X -1  X -1  X -1  X  X
 X  1  1  1  1  0  X  X -1 -1  X  X  X  X  X -1 -1 -1 -1  X
```

Figure 24: Visual explanation of the purpose of the integer index

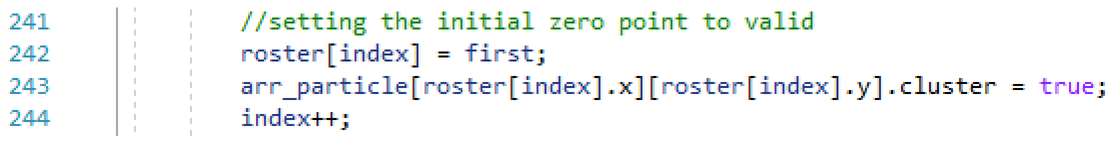


```
236     InitializeRoster(roster, N);
237
238     int index = 0;
239     int seed = 0;
```

Figure 25: Lines 236-239 source2a.c

As you can see in Figure 25, in line **239** an integer seed is created and initialized at 0. To explain the purpose of this value I will elaborate on the purpose and usage of our roster.

Our roster will contain every coordinate of particles contained within our studied cluster. Thus every particle 1 will have it's coordinates defined somewhere in the cluster. This is why in lines **241-244** the initial particles coordinates x and y are added to our cluster.



```
241     //setting the initial zero point to valid
242     roster[index] = first;
243     arr_particle[roster[index].x][roster[index].y].cluster = true;
244     index++;
```

Figure 26: Lines 241-244 source2a.c

The index is added a 1 after this as now every other searched particle will have a jump of 1 or more regarding position 0. If printed, our roster should be equivalent to Figure 27.

Initialized_roster

{-1,-1}
{-1,-1}
…
…
…
…
…
{-1,-1}

Figure 27: Roster after line 247 source2a.c

In line **246** the function AnalyzeRoster, described in lines **133-164** is called.

The method our code employs to “fill in” our rooster with the relevant locations is based on searching every location around the location corresponding to the coordinates roster[seed] and if this location can “connect” with our location roster[seed] to add it to the roster as roster[index] and then add 1 to the value index. Every time a coordinate is added to the ruster it’s corresponding array grid location has it’s cluster Boolean changed to true, which is useful to avoid repetition of searches and for later printing of the grid. Once all locations around roster[seed] are checked for particles with which there might be a connection, the value seed is added 1. And finally once roster[seed] = roster[index] we no longuer have any connected particles (it is a dead end) and our search for coordinates within the cluster stops.

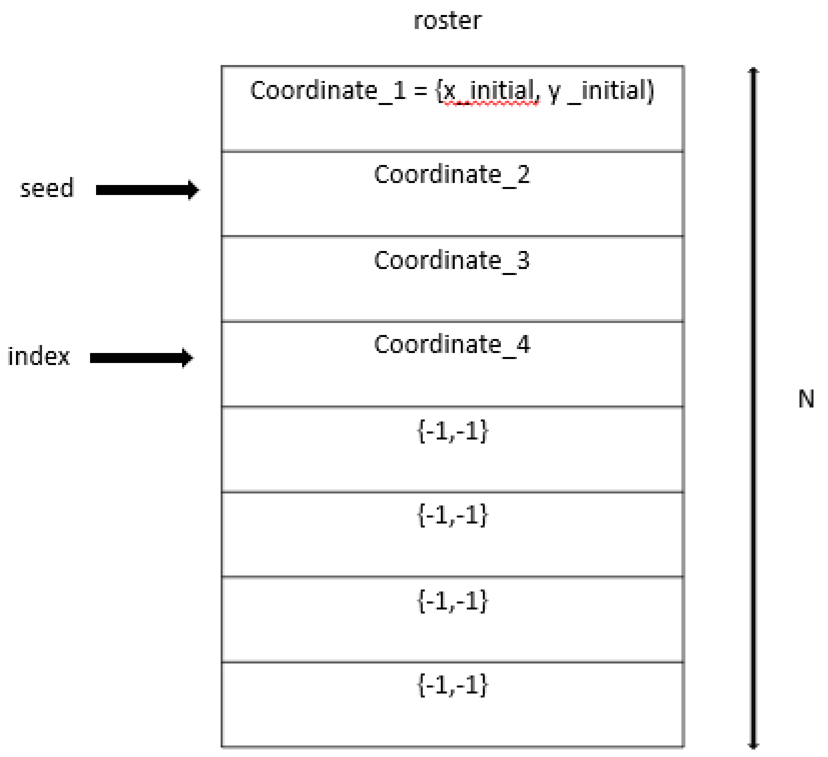


Figure 28: Roster visual representation source2a.c

This is why in the cluster size = N, as there is a maximum on possible N coordinates that could be present within the cluster and thus within our roster.

The function AnalyzeRoster in lines **133-164** describes this process with the use of a while loop that keeps the "search" process going until index = seed. There is also two nested loops that define integer values of d and j ranging from -1 to 1 that describe every location surrounding roster[seed] as represented in Figure 29. Inside both of these loops there is an if loop that ensures that the particle is looking at it's neighboors and not at itself (thus cannot have a j=0 and a d=0).

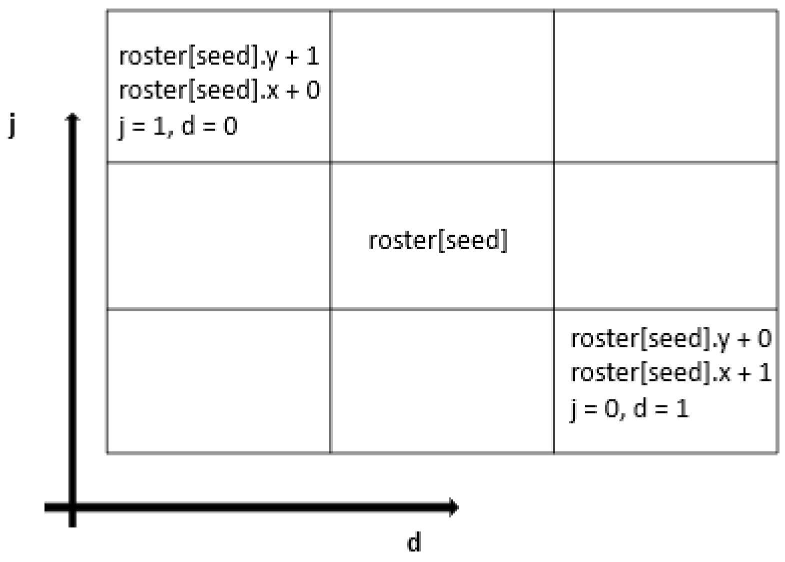
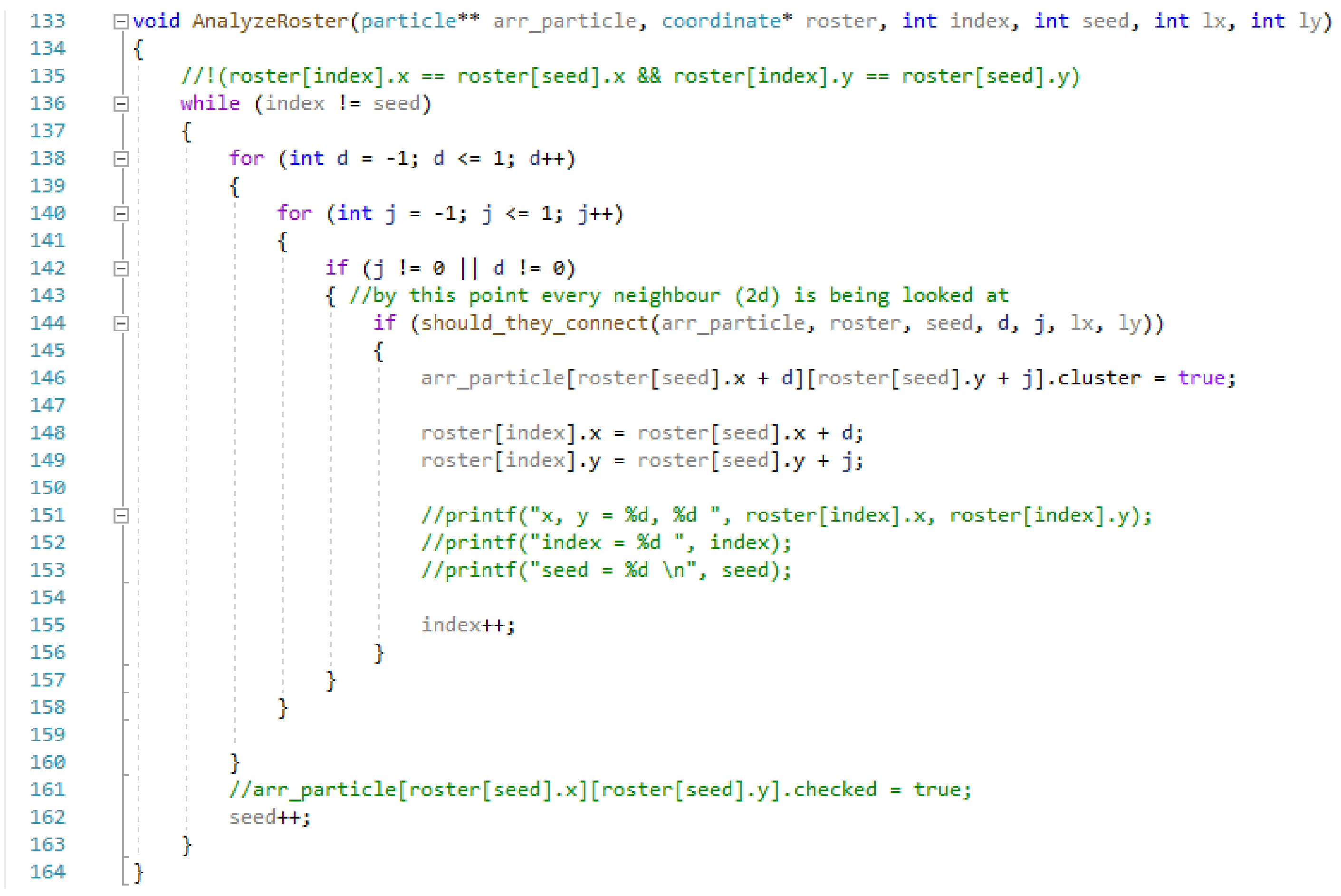


Figure 29: roster[seed].x + d and roster[seed].y + j

Withing these loops there is another if loop that imposes the condition that both the seed and the targeted particle can connect. This is never an issue if the particle are diagonally or vertically separated from each other, yet issues arise if they are diagonally separated from each other. This condition is imposed through the Boolean function should_they_connect defined in lines **107-132**, that also checks that the connection is made between two particles and not between a particle and an empty space.

If this condition is met the coordinates of the location we are checking are added into the roster. This is done for every d and j and once those iterations finish seed is added a one and as long as seed is different from index the cycle restarts.

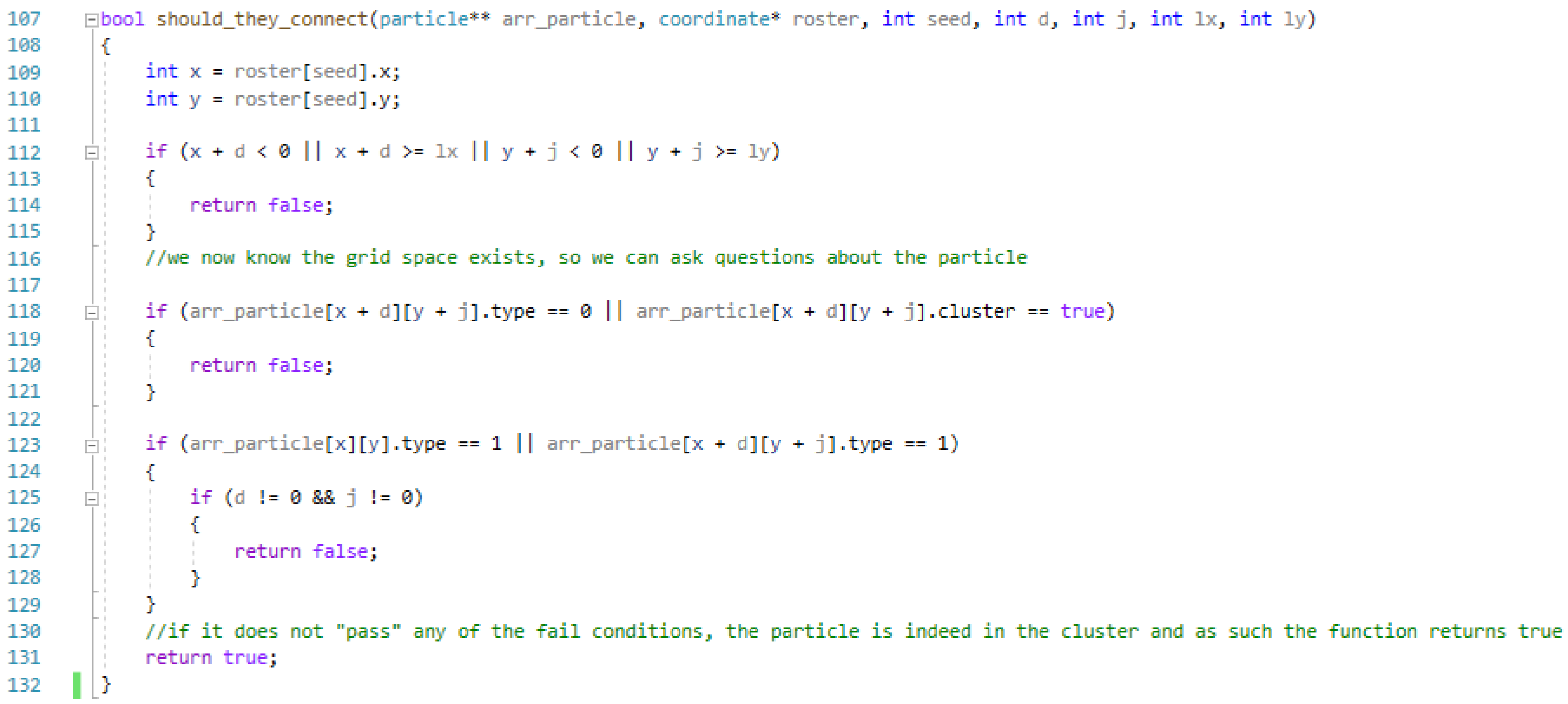


```
133 void AnalyzeRoster(particle** arr_particle, coordinate* roster, int index, int seed, int lx, int ly)
134 {
135     //!(roster[index].x == roster[seed].x && roster[index].y == roster[seed].y)
136     while (index != seed)
137     {
138         for (int d = -1; d <= 1; d++)
139         {
140             for (int j = -1; j <= 1; j++)
141             {
142                 if (j != 0 || d != 0)
143                 { //by this point every neighbour (2d) is being looked at
144                     if (should_they_connect(arr_particle, roster, seed, d, j, lx, ly))
145                     {
146                         arr_particle[roster[seed].x + d][roster[seed].y + j].cluster = true;
147 
148                         roster[index].x = roster[seed].x + d;
149                         roster[index].y = roster[seed].y + j;
150 
151                         //printf("x, y = %d, %d ", roster[index].x, roster[index].y);
152                         //printf("index = %d ", index);
153                         //printf("seed = %d \n", seed);
154 
155                         index++;
156                     }
157                 }
158             }
159 
160         }
161         //arr_particle[roster[seed].x][roster[seed].y].checked = true;
162         seed++;
163     }
164 }
```

Figure 30: Lines 133-164 source2a.c

The function should_they_connect defined in lines **107-132** will return a true Boolean unless proven the countrary. There is 3 if functions that could cause a false return.

1. The first one defined in lines **112-115** ensures that the location we are comparing to our seed is within our grid.
2. The second one defined in lines **118-121** checks that the compared location holds a particle (type different from 0) and that this particle is not already in the cluster and thus not already in the roster so we do not check it more than once.
3. The third loop defined in lines **123-129** deals with issues cause by diagonal connection. If either the seed coordinates or the compared coordinates point towards a particle of type 1 i.e. a non-superconductor, any diagonal connection is “illegal”, and thus if the found connection is indeed illegal (i.e either d=0 or j=0) the function return must be false.

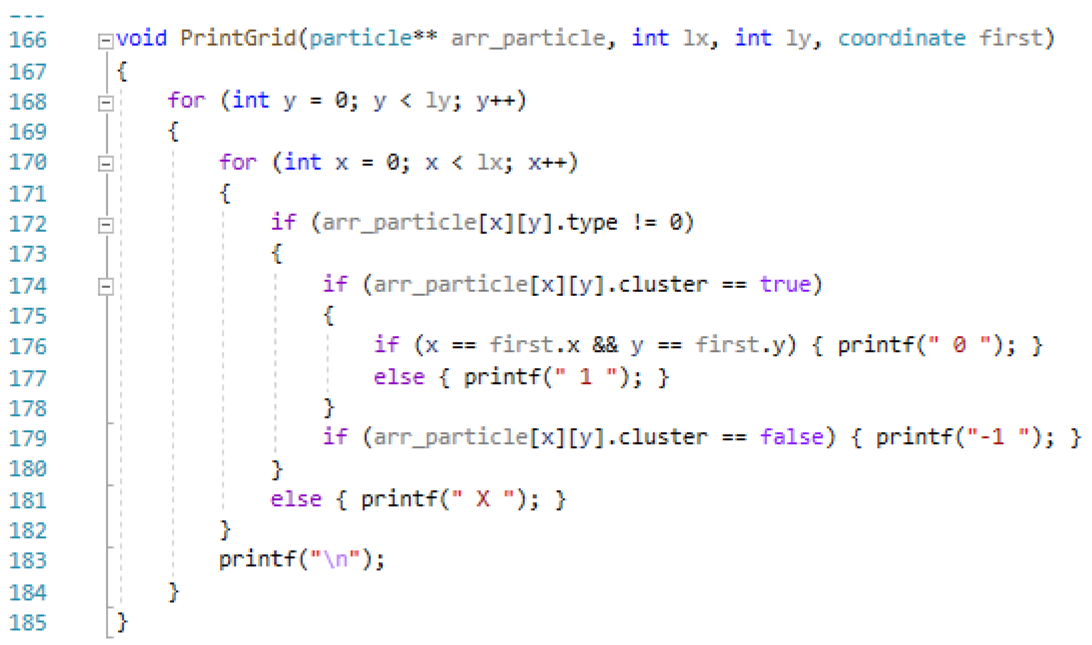


```
107  bool should_they_connect(particle** arr_particle, coordinate* roster, int seed, int d, int j, int lx, int ly)
108  {
109      int x = roster[seed].x;
110      int y = roster[seed].y;
111  
112      if (x + d < 0 || x + d >= lx || y + j < 0 || y + j >= ly)
113      {
114          return false;
115      }
116      //we now know the grid space exists, so we can ask questions about the particle
117  
118      if (arr_particle[x + d][y + j].type == 0 || arr_particle[x + d][y + j].cluster == true)
119      {
120          return false;
121      }
122  
123      if (arr_particle[x][y].type == 1 || arr_particle[x + d][y + j].type == 1)
124      {
125          if (d != 0 && j != 0)
126          {
127              return false;
128          }
129      }
130      //if it does not "pass" any of the fail conditions, the particle is indeed in the cluster and as such the function returns true
131      return true;
132  }
```

Figure 31: Lines 107-132 source2a.c

Within main, following the function AnalyzeRoster, there is a commented PrintRoster function that was used for testing. It is defined (in comments) in lines **99-105**.

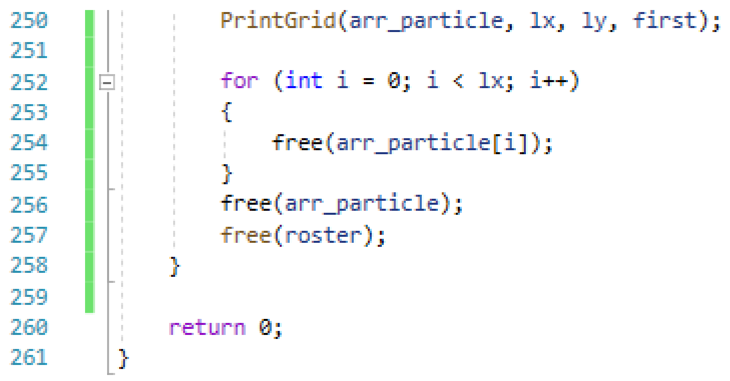
Following this comment the function PrintGrid is called in line **250.** The function PrintGrid is defined in lines **166-185** and iterates through all positions within the array with the use of two nested for loops that iterate through all y and x values. The function checks the locations type, if it is equal to 0 i.e the position is empty it prints a “ X ” symbol. If it has a non-zero type it checks if it is contained within the cluster. If it isn’t contained within the cluster it prints a “ -1 “ symbol. If it is contained within the cluster it checks wherever it is position 0 or not. I fit is position 0 it prints a “ 0 “ symbol and if it isn’t then it prints a “ 1 “ symbol.



```
---
166  void PrintGrid(particle** arr_particle, int lx, int ly, coordinate first)
167  {
168      for (int y = 0; y < ly; y++)
169      {
170          for (int x = 0; x < lx; x++)
171          {
172              if (arr_particle[x][y].type != 0)
173              {
174                  if (arr_particle[x][y].cluster == true)
175                  {
176                      if (x == first.x && y == first.y) { printf(" 0 "); }
177                      else { printf(" 1 "); }
178                  }
179                  if (arr_particle[x][y].cluster == false) { printf("-1 "); }
180              }
181              else { printf(" X "); }
182          }
183          printf("\n");
184      }
185  }
```

Figure 32: Lines 166-185 source2a.c

Finally in n lines **252-257** all created malloc pointers are freed, and the main function ends in line **261**.



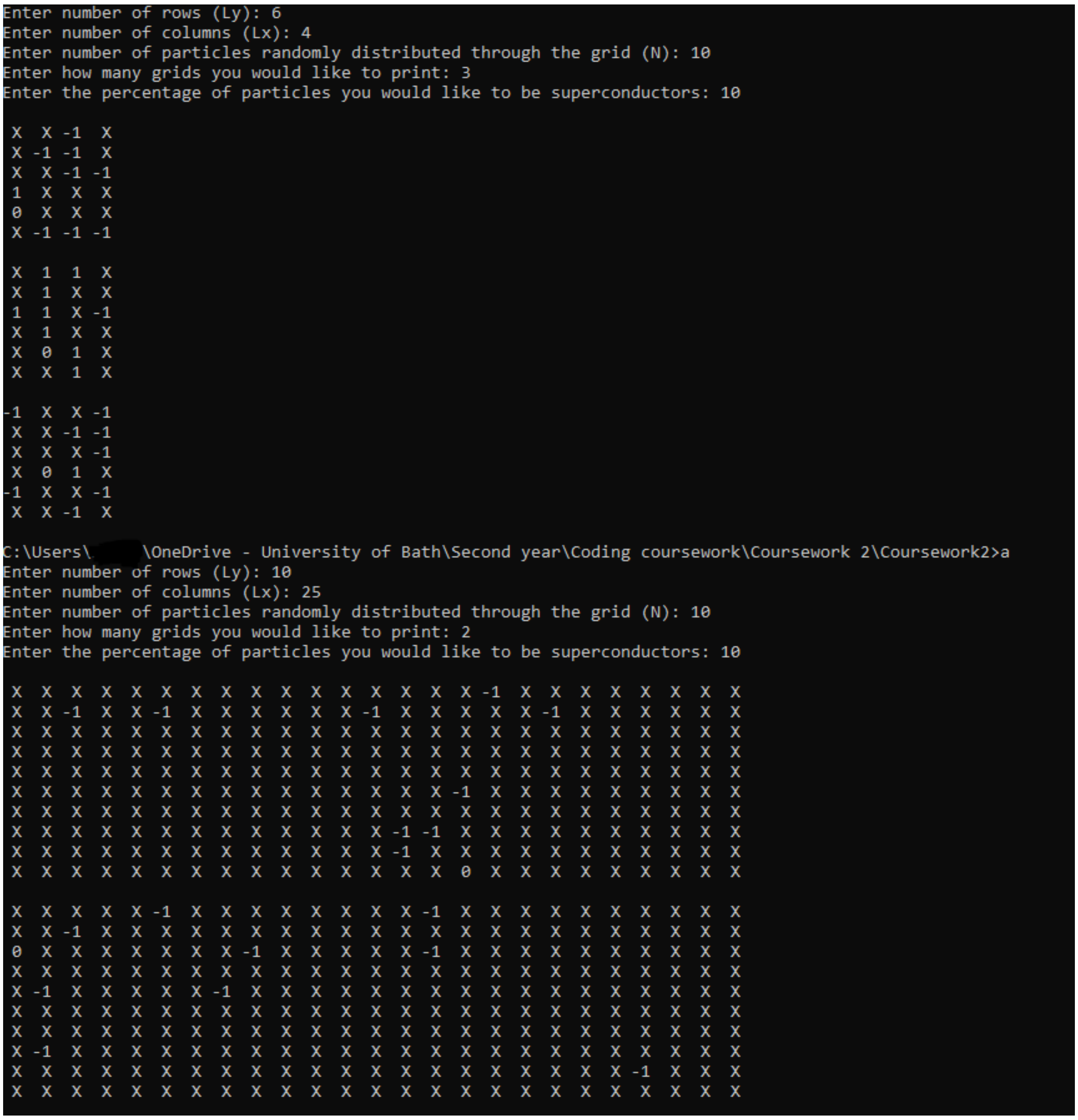
```
250          PrintGrid(arr_particle, lx, ly, first);
251
252          for (int i = 0; i < lx; i++)
253          {
254              free(arr_particle[i]);
255          }
256          free(arr_particle);
257          free(roster);
258      }
259
260      return 0;
261  }
```

Figure 33: Lines 250-261 source2a.c

Results

As seen in Figure 4 the particles connected to the initial particle as well as all the other properties regarding the inputted values are correct.

We thus can claim that the program source2a.c correctly answers questions 2.a)



```
Enter number of rows (Ly): 6
Enter number of columns (Lx): 4
Enter number of particles randomly distributed through the grid (N): 10
Enter how many grids you would like to print: 3
Enter the percentage of particles you would like to be superconductors: 10

 X  X -1  X
 X -1 -1  X
 X  X -1 -1
 1  X  X  X
 0  X  X  X
 X -1 -1 -1

 X  1  1  X
 X  1  X  X
 1  1  X -1
 X  1  X  X
 X  0  1  X
 X  X  1  X

-1  X  X -1
 X  X -1 -1
 X  X  X -1
 X  0  1  X
-1  X  X -1
 X  X -1  X

C:\Users\[redacted]\OneDrive - University of Bath\Second year\Coding coursework\Coursework 2\Coursework2>a
Enter number of rows (Ly): 10
Enter number of columns (Lx): 25
Enter number of particles randomly distributed through the grid (N): 10
Enter how many grids you would like to print: 2
Enter the percentage of particles you would like to be superconductors: 10

 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X -1  X  X  X  X  X  X  X  X
 X  X -1  X  X -1  X  X  X  X  X  X -1  X  X  X  X  X -1  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X -1  X  X  X  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X -1 -1  X  X  X  X  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X -1  X  X  X  X  X  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  0  X  X  X  X  X  X  X  X  X

 X  X  X  X  X -1  X  X  X  X  X  X  X  X -1  X  X  X  X  X  X  X  X  X  X
 X  X -1  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
 0  X  X  X  X  X  X  X -1  X  X  X  X  X -1  X  X  X  X  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
 X -1  X  X  X  X  X -1  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
 X -1  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X -1  X  X  X  X
 X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
```

Figure 34: Results obtained when compiling the program source2a.c

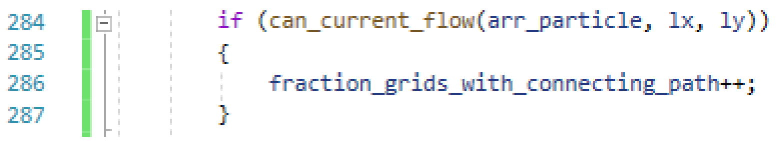
2 .b: source2b.c program

Design presentation and discussion

Question 2.b) asks us to recognise whether the found cluster contains a path between the two electrodes and to output the fraction of grids containing a path with respect to the number of printed grids.

A simple way to recognize wherever this is the case or not is to check wherever there is a 1 particle present in the top and bottom rows (as these are the ones connected to our electrodes). For this purpose we write the function can_current_flow, defined in lines **187-208**. The function checks wherever there is any position in the first row (y=0) with a cluster = true. It does this through a for loop that iterates through all possible x positions. If there isn't one the function returns false, but if there is one it then repeats the process for the last row (y=ly-1). If this is true it returns true.

Within our main that is defined in lines **212-302**, before our for loop that iterates through every grid, we create an integer value fraction_grids_with_connecting_path = 0, that will grow by one everytime a grid returns a true can_current_flow function. This is checked with an if loop in lines **284-287** (thus within the iterating for loop).



```
284         if (can_current_flow(arr_particle, lx, ly))
285         {
286             fraction_grids_with_connecting_path++;
287         }
```

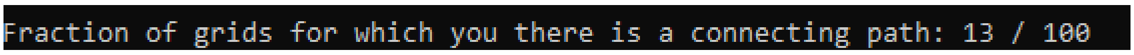
Figure 35: Lines 284-287 source2b.c

After the end of our grid loop, in line **299** ,we print our fraction_grids_with_connecting_path, a " / " symbol and our n_grids value, to thus obtain the fraction of grids for there is a connecting path in our command line.

Results

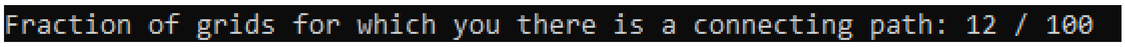
For speed purposes we comment all printing lines in this code except line **299**.

We obtain results in the range of 10<x<20 as seen in Figure 36-39



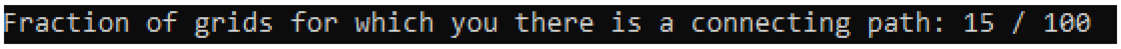
```
Fraction of grids for which you there is a connecting path: 13 / 100
```

Figure 36: Results obtained when compiling the program source2b.c (1)



```
Fraction of grids for which you there is a connecting path: 12 / 100
```

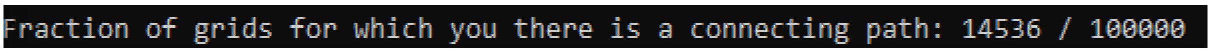
Figure 37: Results obtained when compiling the program source2b.c (2)



```
Fraction of grids for which you there is a connecting path: 15 / 100
```

Figure 38: Results obtained when compiling the program source2b.c (3)

If we run the program with the same grid properties (Lx = 6, Ly = 4, N = 10 and f_{SC} = 10%) 100000 times we obtain 14536/100000 which lets us claim that over 100 grids we should b finding results relatively close to 14, which has been the case in the previous figures.

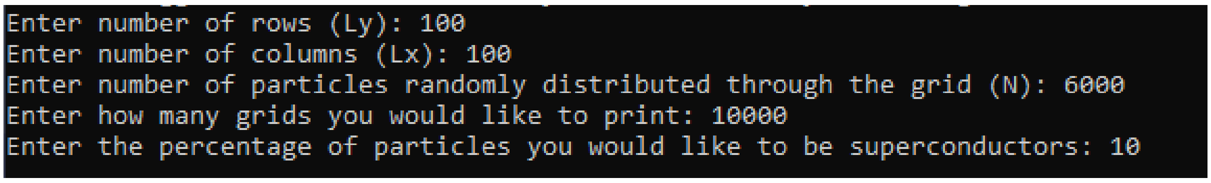


```
Fraction of grids for which you there is a connecting path: 14536 / 100000
```

Figure 39: Results obtained when compiling the program source2b.c over 100000 grids

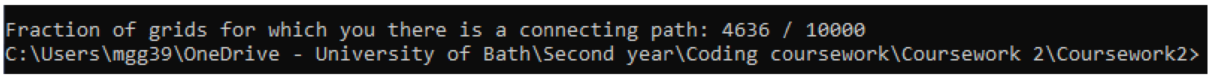
2 .c: source2b.c program

When we run our program source2b.c with the properties Lx = 100, Ly = 100, with N = 6000 and f_{SC} = 10% we obtain Figures 40-43.
As seen in Figure 43 if we run the program with input Lx = 100, Ly = 100, with N = 6000 and f_{SC} = 10%, for 100000 grids we obtain a fraction of 46846/100000, which given the number of grid iterations has what we could consider a negligible error bar.



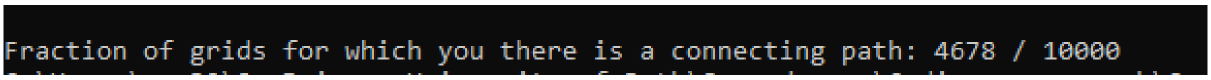
```
Enter number of rows (Ly): 100
Enter number of columns (Lx): 100
Enter number of particles randomly distributed through the grid (N): 6000
Enter how many grids you would like to print: 10000
Enter the percentage of particles you would like to be superconductors: 10
```

Figure 40: Results obtained when compiling the program source2b.c over 10000 grids (1)



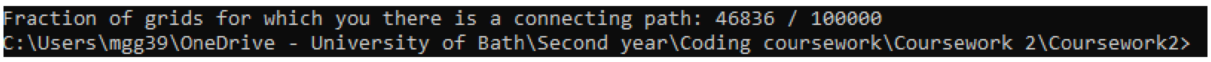
```
Fraction of grids for which you there is a connecting path: 4636 / 10000
C:\Users\mgg39\OneDrive - University of Bath\Second year\Coding coursework\Coursework 2\Coursework2>
```

Figure 41: Results obtained when compiling the program source2b.c over 10000 grids (2)



```
Fraction of grids for which you there is a connecting path: 4678 / 10000
```

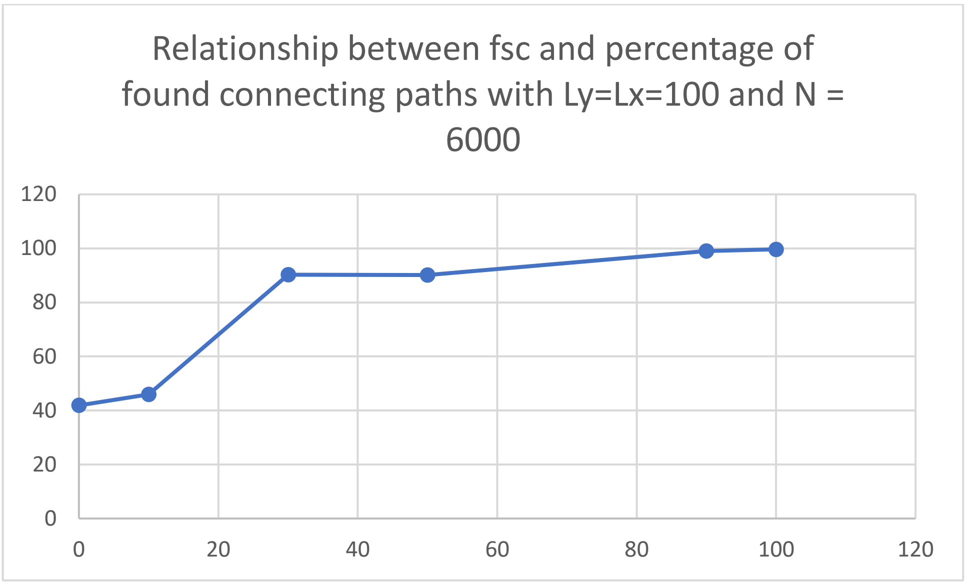
Figure 42: Results obtained when compiling the program source2b.c over 10000 grids (3)



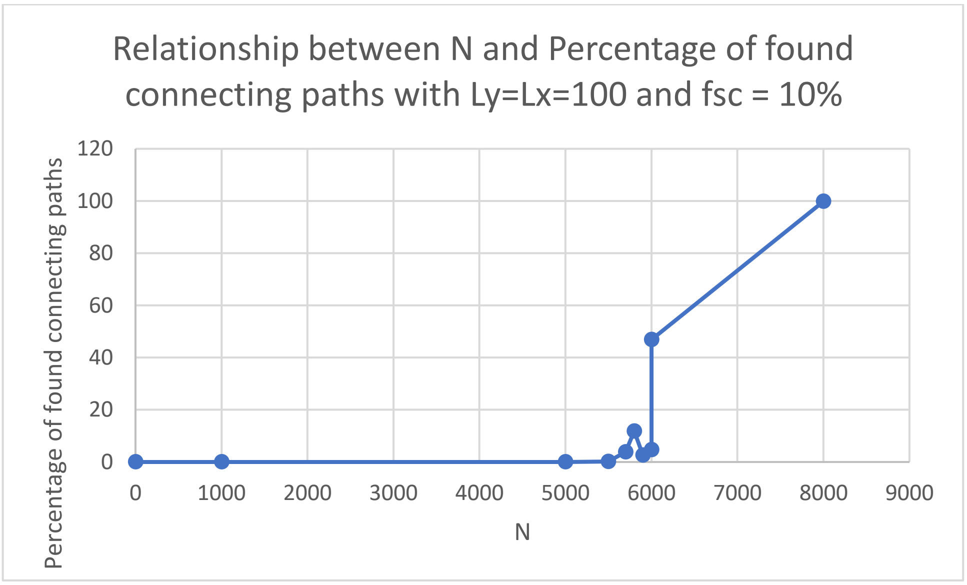
```
Fraction of grids for which you there is a connecting path: 46836 / 100000
C:\Users\mgg39\OneDrive - University of Bath\Second year\Coding coursework\Coursework 2\Coursework2>
```

Figure 43: Results obtained when compiling the program source2b.c over 100000 grids

If we maintain Lx=Ly=100 and N=6000, we can observe the following relationship between fsc and the percentage of found connecting paths (every data point in this grid has been compared to 10^4 grids, and thus has an accuracy of such):



If we maintain Lx=Ly=100 and fsc=10%, we can observe the following relationship between N and the percentage of found connecting paths (every data point in this grid has been compared to 10^4 grids, and thus has an accuracy of such):



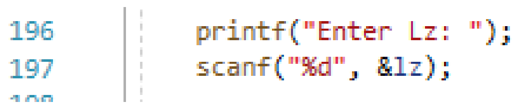
We obtain an unexpected peak at N=6000.

<u>Question 3: source3.c program</u>

<u>Design presentation and discussion</u>

Program source3.c is relatively similar to program source2b.c, we simply add an extra dimension which is translated by x,y -> x,y,z and lx, ly -> lx, ly, lz. For the purpose of explaining how this code thus differs from source2b.c I will go through all found differences in order of appearance within the main define din lines **186-278**.

Our first difference is the input of a variable ly in lines **196-197**.

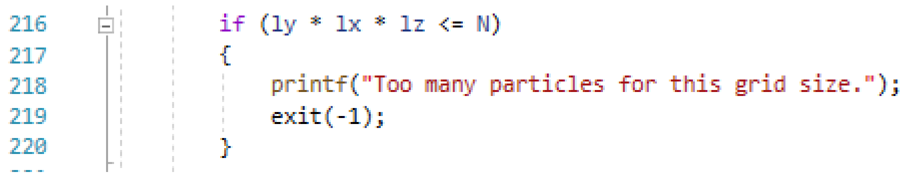


```
196     printf("Enter Lz: ");
197     scanf("%d", &lz);
```

Figure 44: Lines 196-197 source3.c

Similarly to lx and ly, lx will be used in many of our functions, as it represents a third z axis.

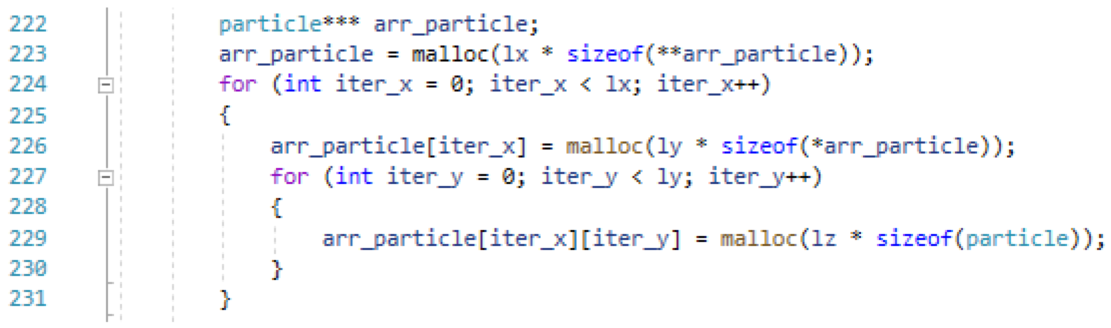
The first use of lz is found in lines **216-220** as now the size of the grid is lx*ly*lz and thus N must be inferior or equal to lx*ly*lz instead of just lx*ly.



```
216     if (ly * lx * lz <= N)
217     {
218         printf("Too many particles for this grid size.");
219         exit(-1);
220     }
```

Figure 45: Lines 216-220 source3.c

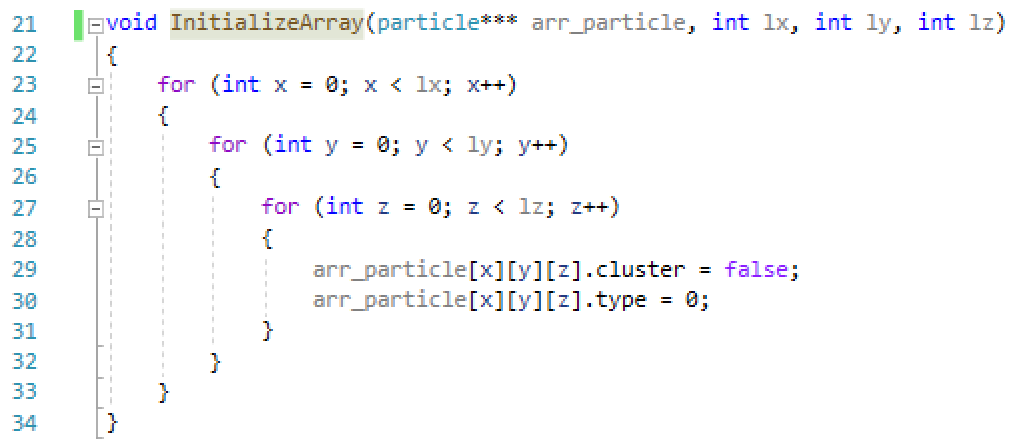
Our next difference in comparison to our source2b.c program is our arr_particle definition. It is now a 3D array pointer and is defined accordingly.



```
222     particle*** arr_particle;
223     arr_particle = malloc(lx * sizeof(**arr_particle));
224     for (int iter_x = 0; iter_x < lx; iter_x++)
225     {
226         arr_particle[iter_x] = malloc(ly * sizeof(*arr_particle));
227         for (int iter_y = 0; iter_y < ly; iter_y++)
228         {
229             arr_particle[iter_x][iter_y] = malloc(lz * sizeof(particle));
230         }
231     }
```

Figure 46: Lines 222-231 source3.c

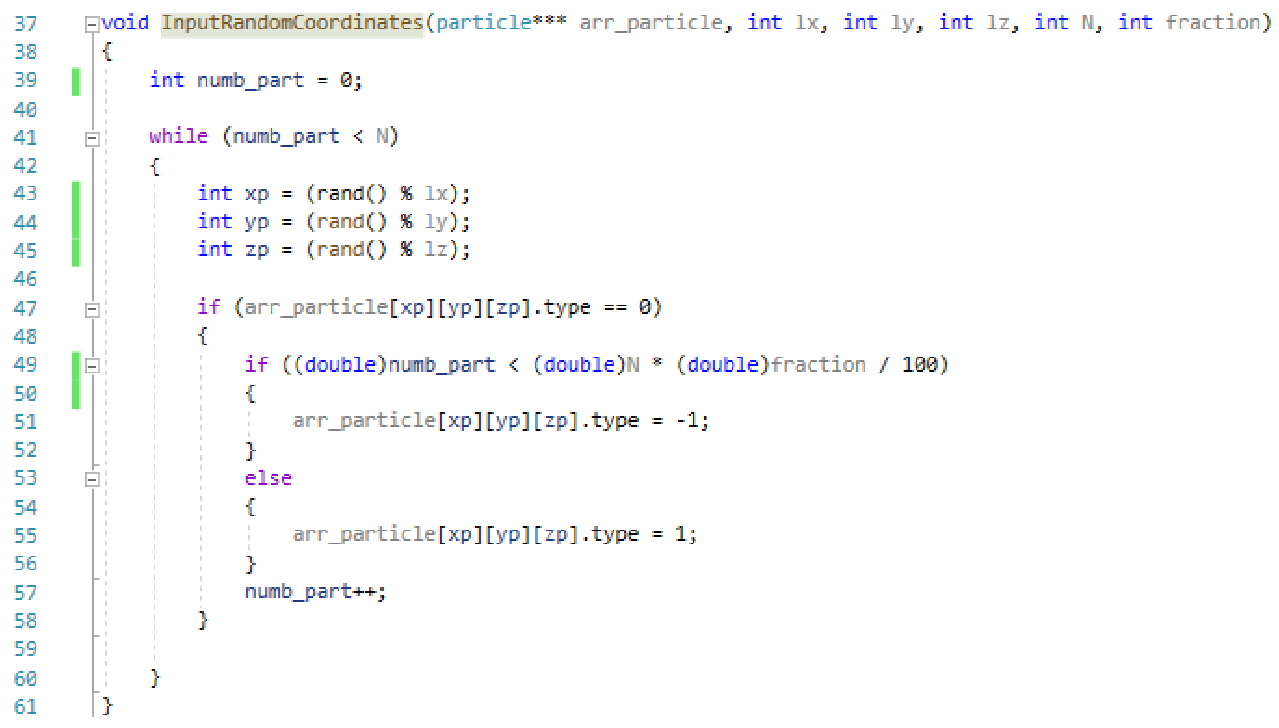
Lz is also present in our InitializeArray function, defined in lines **21-34**. It cause an additional nested loop that iterates through the values of z.



```
21  void InitializeArray(particle*** arr_particle, int lx, int ly, int lz)
22  {
23      for (int x = 0; x < lx; x++)
24      {
25          for (int y = 0; y < ly; y++)
26          {
27              for (int z = 0; z < lz; z++)
28              {
29                  arr_particle[x][y][z].cluster = false;
30                  arr_particle[x][y][z].type = 0;
31              }
32          }
33      }
34  }
```

Figure 47: Lines 21-34 source3.c

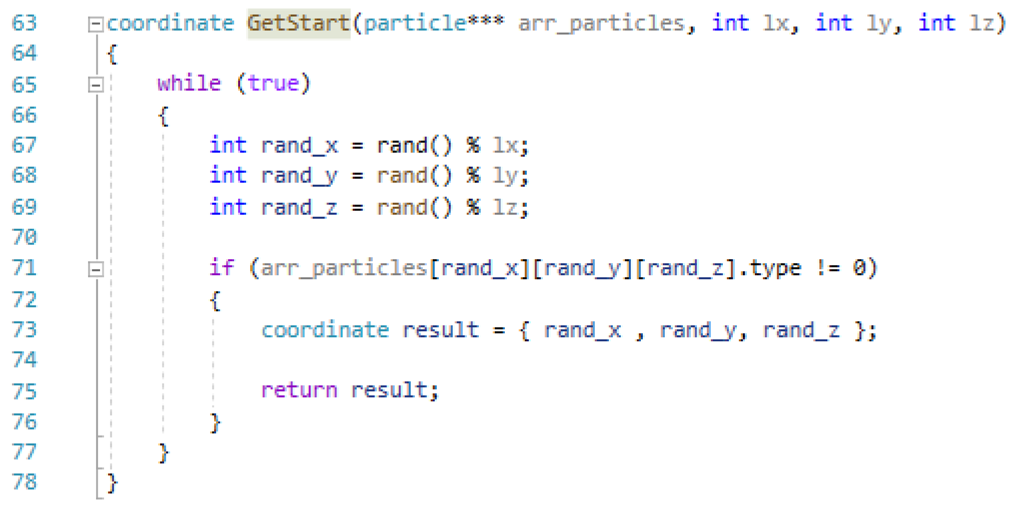
It similarly affects function InputRandomCoordinates defined in lines **37-61**, as this one contains an extra random z number zp that is placed in the 3rd dimension of arr_particle.



```
37  void InputRandomCoordinates(particle*** arr_particle, int lx, int ly, int lz, int N, int fraction)
38  {
39      int numb_part = 0;
40
41      while (numb_part < N)
42      {
43          int xp = (rand() % lx);
44          int yp = (rand() % ly);
45          int zp = (rand() % lz);
46
47          if (arr_particle[xp][yp][zp].type == 0)
48          {
49              if ((double)numb_part < (double)N * (double)fraction / 100)
50              {
51                  arr_particle[xp][yp][zp].type = -1;
52              }
53              else
54              {
55                  arr_particle[xp][yp][zp].type = 1;
56              }
57              numb_part++;
58          }
59
60      }
61  }
```

Figure 48: Lines 37-61 source3.c

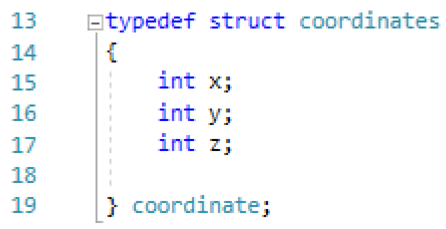
The initial coordinate must be also defined by a z position, thus similarly to InputRandomCoordinates the GetStart function, defined in lines **63-78** must find a random z location.



```c
63  coordinate GetStart(particle*** arr_particles, int lx, int ly, int lz)
64  {
65      while (true)
66      {
67          int rand_x = rand() % lx;
68          int rand_y = rand() % ly;
69          int rand_z = rand() % lz;
70
71          if (arr_particles[rand_x][rand_y][rand_z].type != 0)
72          {
73              coordinate result = { rand_x , rand_y, rand_z };
74
75              return result;
76          }
77      }
78  }
```

Figure 49: Lines 63-78 source3.c

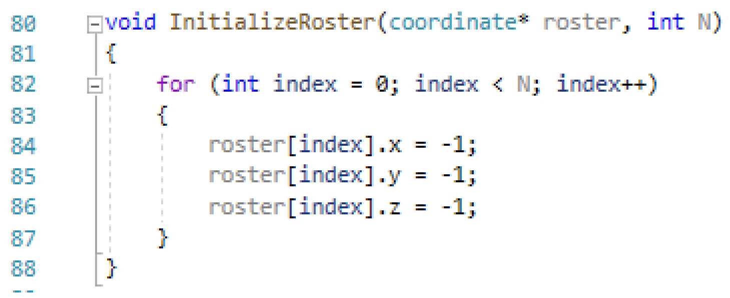
As the roster is a 1 dimensional array we do not need to change it, yet we must change the definition of a coordinate structure, lines **13-19**, to accommodate for a z coordinate value.



```c
13  typedef struct coordinates
14  {
15      int x;
16      int y;
17      int z;
18
19  } coordinate;
```

Figure 50: Lines 13-19 source3.c

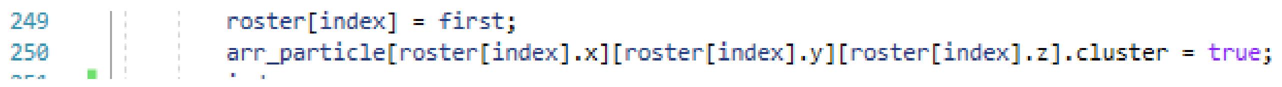
Following this change the function InitializerRoster, defined in lines **80-88**, must also initialize the integers z with a value (for consistency we choose -1).



```c
80  void InitializeRoster(coordinate* roster, int N)
81  {
82      for (int index = 0; index < N; index++)
83      {
84          roster[index].x = -1;
85          roster[index].y = -1;
86          roster[index].z = -1;
87      }
88  }
```

Figure 51: Lines 80-88 source3.c

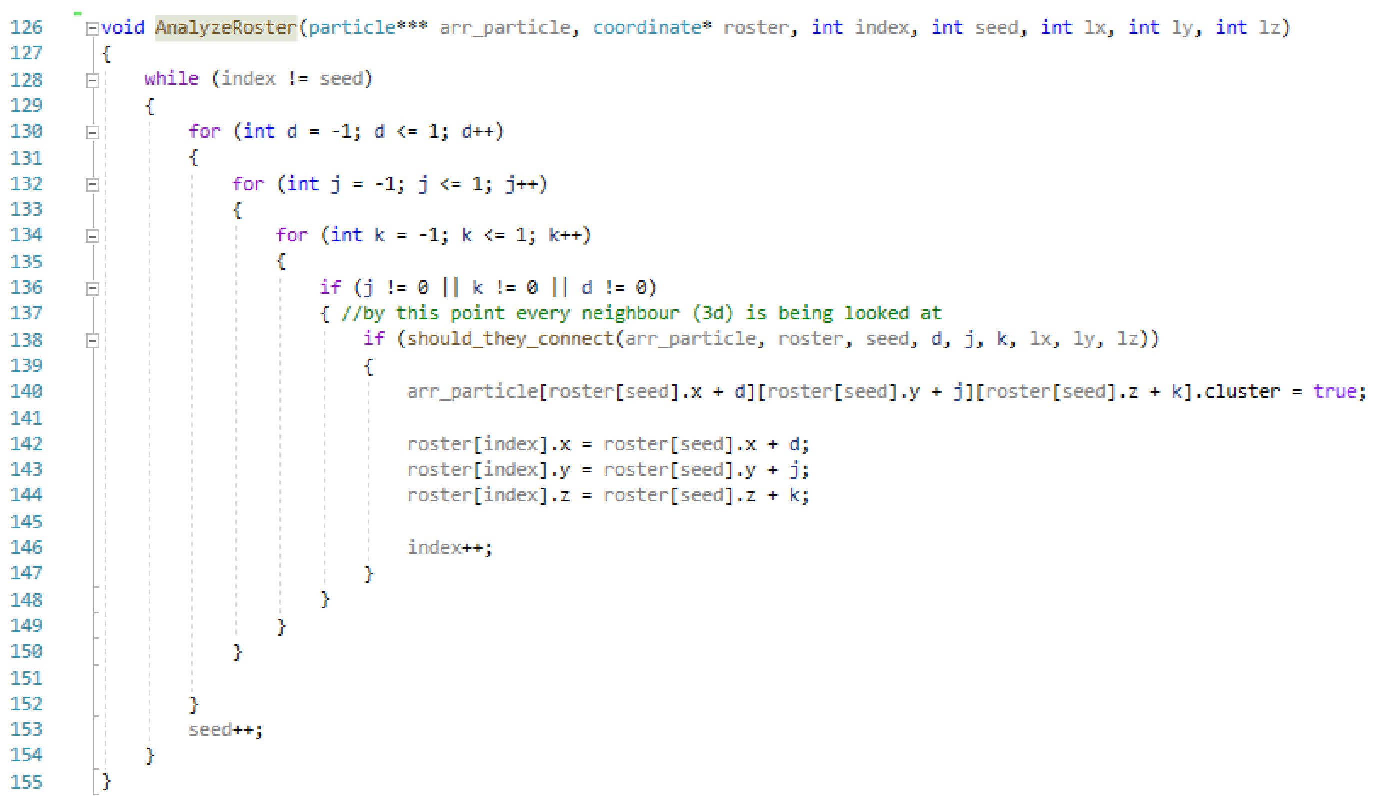
The initial particle is then added to the roster in line **249** and line **250** gives it's arr_particle location a cluster = true, while taking into account the 3rd dimension z.



```
249            roster[index] = first;
250            arr_particle[roster[index].x][roster[index].y][roster[index].z].cluster = true;
```

Figure 52: Lines 249-250 source3.c

Then function AnalyzeRoster, called in line **253** and defined in lines **126-155**, is also modified due to the 3rd dimension. On top of the d and j for loop there is an additional k loop emplid to loop through the z neighboors. This causes an additional or condition for the particle not to be looking at itself, in line **136**. In addition to this it will also affect function should_they_connect.



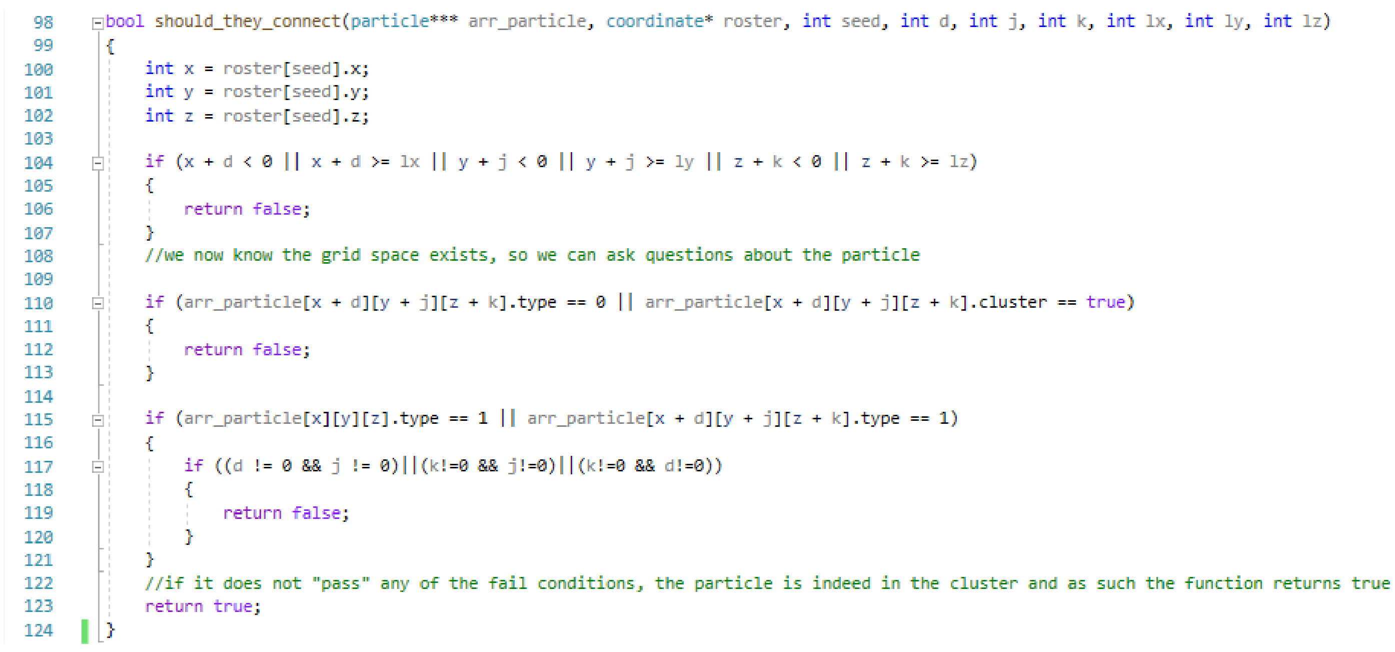
```
126  void AnalyzeRoster(particle*** arr_particle, coordinate* roster, int index, int seed, int lx, int ly, int lz)
127  {
128      while (index != seed)
129      {
130          for (int d = -1; d <= 1; d++)
131          {
132              for (int j = -1; j <= 1; j++)
133              {
134                  for (int k = -1; k <= 1; k++)
135                  {
136                      if (j != 0 || k != 0 || d != 0)
137                      { //by this point every neighbour (3d) is being looked at
138                          if (should_they_connect(arr_particle, roster, seed, d, j, k, lx, ly, lz))
139                          {
140                              arr_particle[roster[seed].x + d][roster[seed].y + j][roster[seed].z + k].cluster = true;
141
142                              roster[index].x = roster[seed].x + d;
143                              roster[index].y = roster[seed].y + j;
144                              roster[index].z = roster[seed].z + k;
145
146                              index++;
147                          }
148                      }
149                  }
150              }
151
152          }
153          seed++;
154      }
155  }
```

Figure 53: Lines 126-155 source3.c

In the function should_they_connect the 3rd dimension affects every if loop differently. I will thus explain them in order of appearance.

1. For the first if loop, lines **104-107**, these is an additional || condition for z to be within it's range.

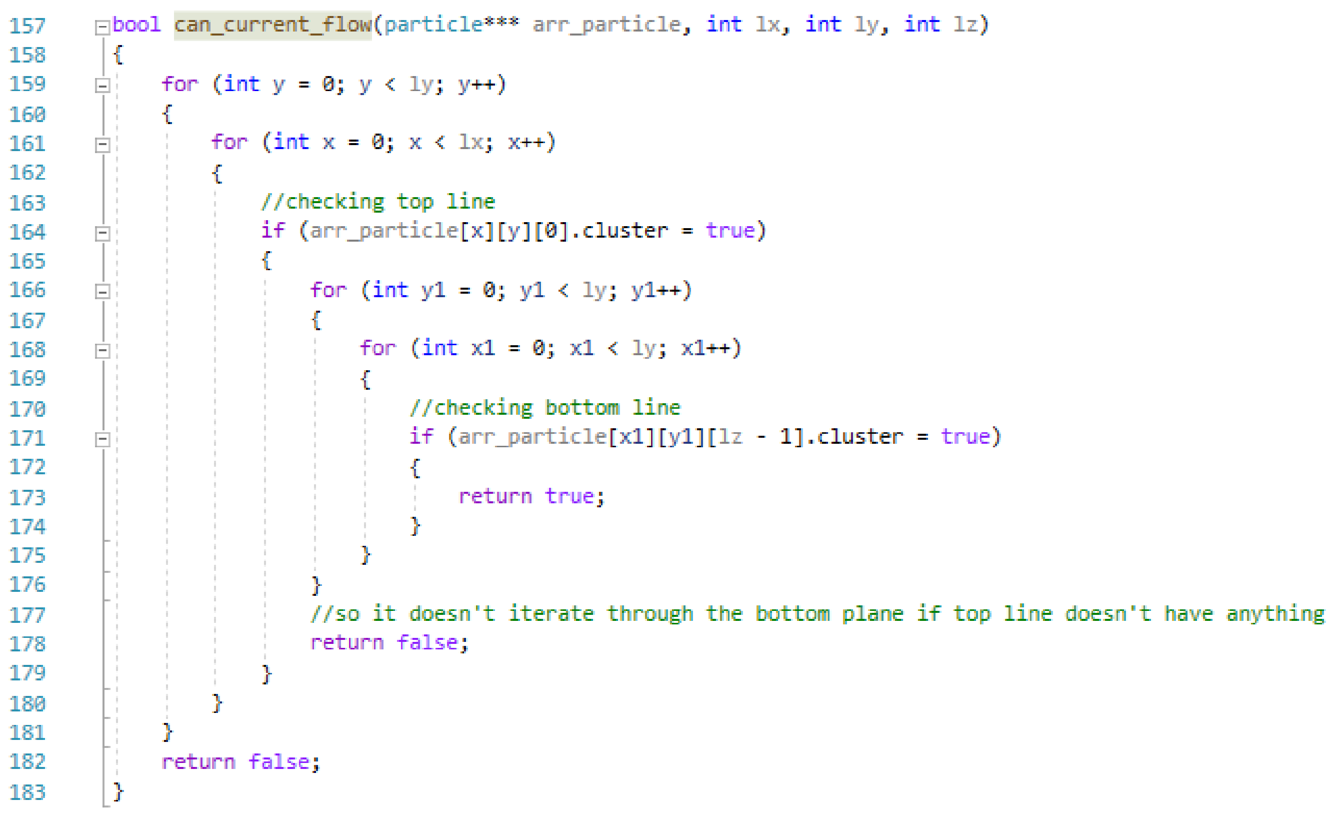
2. For the second loop, lines **110-113**, we simply add the third dimension to our arr_particle description.
3. For our final loop, lines **115-121**, similarly to our second if loop we simply add the third dimension to our arr_particle description. But for the nested loop within this loop we need to add two additional condition, as if two or more values regarding j, k and d, are equal to 0 the non diagonal condition is no longer met.



```
98   bool should_they_connect(particle*** arr_particle, coordinate* roster, int seed, int d, int j, int k, int lx, int ly, int lz)
99   {
100      int x = roster[seed].x;
101      int y = roster[seed].y;
102      int z = roster[seed].z;
103  
104      if (x + d < 0 || x + d >= lx || y + j < 0 || y + j >= ly || z + k < 0 || z + k >= lz)
105      {
106          return false;
107      }
108      //we now know the grid space exists, so we can ask questions about the particle
109  
110      if (arr_particle[x + d][y + j][z + k].type == 0 || arr_particle[x + d][y + j][z + k].cluster == true)
111      {
112          return false;
113      }
114  
115      if (arr_particle[x][y][z].type == 1 || arr_particle[x + d][y + j][z + k].type == 1)
116      {
117          if ((d != 0 && j != 0)||(k!=0 && j!=0)||(k!=0 && d!=0))
118          {
119              return false;
120          }
121      }
122      //if it does not "pass" any of the fail conditions, the particle is indeed in the cluster and as such the function returns true
123      return true;
124  }
```

Figure 54: Lines 98-124 source3.c

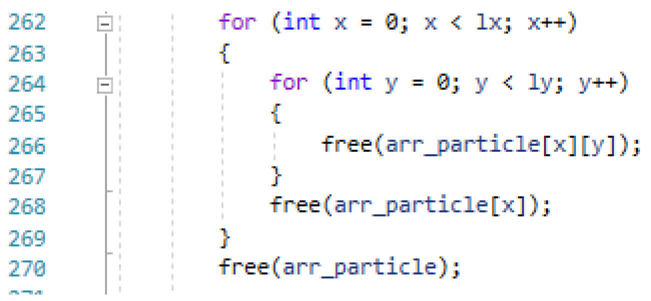
In our main in lines **255-258** we find the can_current_flow function. This function, defined in lines **157-183**, is also modified accordingly to the presence of a 3rd dimension. As we now check the bottom and top z lines, while iterating through every columns value x and then every row value y. Apart from this change in "priority" from y to z, the function works similarly to it's 2 dimensional counterpart.



```c
157 bool can_current_flow(particle*** arr_particle, int lx, int ly, int lz)
158 {
159     for (int y = 0; y < ly; y++)
160     {
161         for (int x = 0; x < lx; x++)
162         {
163             //checking top line
164             if (arr_particle[x][y][0].cluster = true)
165             {
166                 for (int y1 = 0; y1 < ly; y1++)
167                 {
168                     for (int x1 = 0; x1 < ly; x1++)
169                     {
170                         //checking bottom line
171                         if (arr_particle[x1][y1][lz - 1].cluster = true)
172                         {
173                             return true;
174                         }
175                     }
176                 }
177                 //so it doesn't iterate through the bottom plane if top line doesn't have anything
178                 return false;
179             }
180         }
181     }
182     return false;
183 }
```

Figure 55: Lines 157-183 source3.c

Finally at the end of our main the only changing lines is the ones describing the freeing of the arr_particle malloc, which is now done in 3D instead of 2D. This occurs in lines **261-271**. We start by freeing the z dimension, followed by the y dimension and finally free the x dimension in line **270**.



```c
262         for (int x = 0; x < lx; x++)
263         {
264             for (int y = 0; y < ly; y++)
265             {
266                 free(arr_particle[x][y]);
267             }
268             free(arr_particle[x]);
269         }
270         free(arr_particle);
```

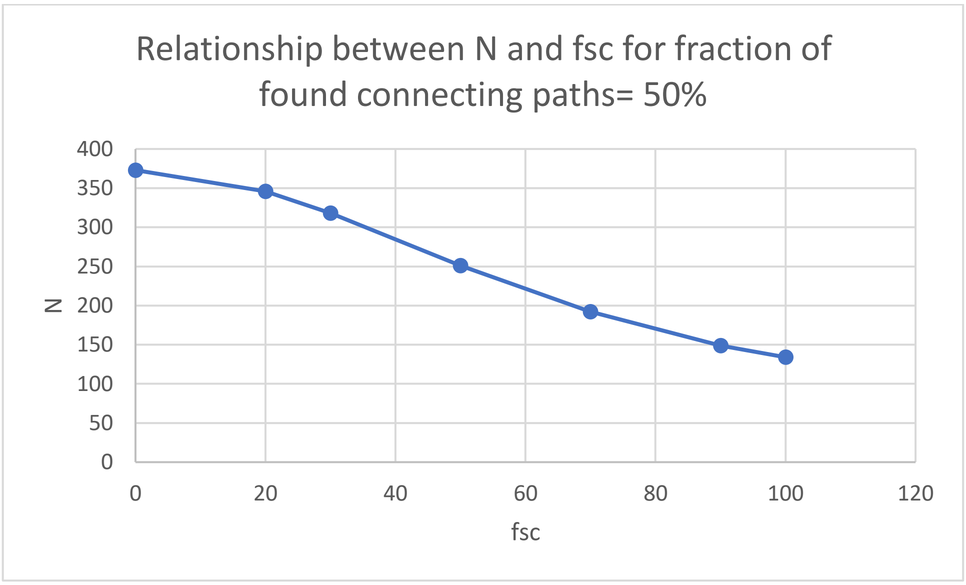
Figure 56: Lines 262-270 source3.c

Results

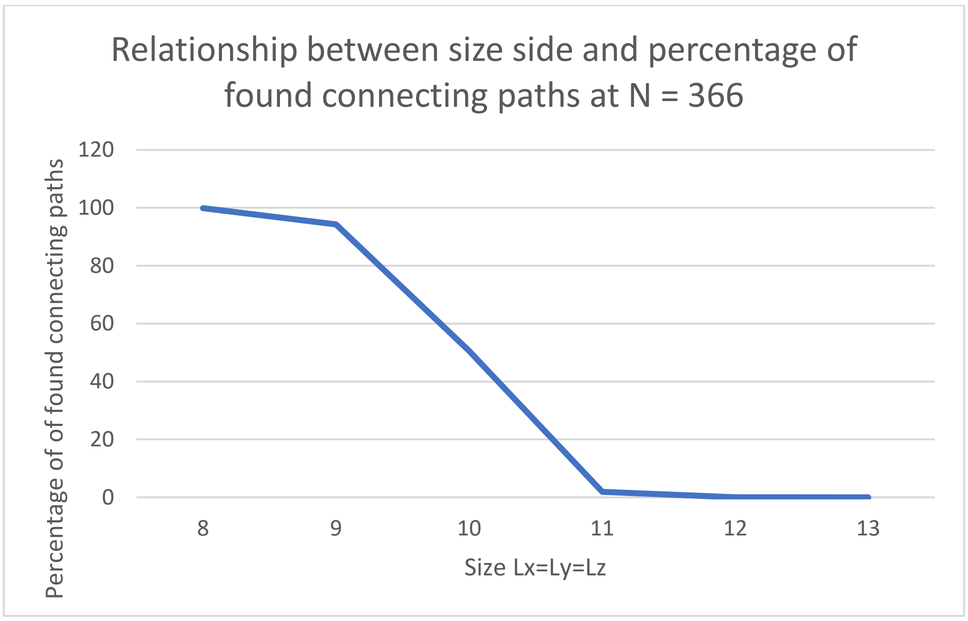
In this section we assume we run the program with properties Lx = Ly = Lz = 10 and f_{sc}=10%, unless stated otherwise.

After some trial and error we find that the smallest N we can obtain in order to find a connecting path 50% of the time is N=366. As through 10000 iterations N=366 finds a fraction of 50542/ 100000 and N=365 finds a fraction of 49937 / 100000. Given the large number of grid iterations we assume the error bar is negligible.

If we repeat this process with different N value we obtain the following fsc and N relation when finding fraction = 50% (every data point in this grid has been compared to 10^5 grids, and thus has an accuracy of such):



If we repeat this process with the same N value (i.e 10%) but different side (lx=ly=lz) values we obtain the following side size and fraction (in percentage) relation (every data point in this grid has been compared to 10^5 grids, and thus has an accuracy of such):


Relationship between size side and percentage of found connecting paths at N = 366
Percentage of of found connecting paths
120
100
80
60
40
20
0
8
9
10
11
12
13
Size Lx=Ly=Lz